HEWLETT
PACKARD

Symbolic Debug/1000

User's Manual

**Software Technology Division
11000 Wolfe Road
Cupertino, CA 95014-9804**

Manual Part No. 92860-90001
E1193

Printed in U.S.A. November 1993
Seventh Edition

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARs 252.227.7013.

Printing History

The Printing History below identifies the edition of this manual and any updates that are included. Periodically, update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this printing history page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past updates; however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all updates.

To determine what manual edition and update is compatible with your current software revision code, refer to the Manual Numbering File or the Computer User's Documentation Index. (The Manual Numbering File is included with your software. It consists of an "M" followed by a five digit product number.)

First Edition	Jun 1979	
Update 1	Jun 1983	
Second Edition	Jun 1984	
Third Edition	Jan 1986	
Fourth Edition	Aug 1987	Rev. 5000 (Software Update 5.0)
Update 1	Jul 1990	Rev. 5020 (Software Update 5.2)
Fifth Edition	Nov 1991	C/1000 Information Incorporated
Sixth Edition	Dec 1992	Rev. 6000, Xdb Information added
Seventh Edition	Nov 1993	Rev. 6100 (Software Update 6.1)

Preface

This manual is a tutorial guide and reference for the Symbolic Debug/1000 program. It is assumed that you know the rules of the programming language being used and know how to edit, compile, and link programs on an RTE system.

Chapter 1 Outlines what the Symbolic Debug/1000 program can do. This chapter describes the features of Debug and introduces the conventions used in this manual.

Chapter 2 Gives installation and configuration information. Describes the steps necessary to prepare a program for debugging, which includes debug compiler and link options. This chapter also describes the Debug runstring options.

Chapter 3 Describes the debugging rules and how to use the Debug program. Various examples are used to illustrate common Debug functions.

Chapter 4 Describes locations, arrays, and expressions and how they are used.

Chapter 5 Contains detailed descriptions of all the Debug commands. This chapter can be used as a reference section.

Chapter 6 Describes profiling, a special feature of the Debug program. Descriptions of the profiling commands also are included.

Chapter 7 Describes the Xdb compatibility mode of Debug/1000. Gives the Xdb runstring, runstring options, and Xdb commands.

Appendix A Contains a list of error messages and explanations.

Table of Contents

Chapter 1
Introduction to Debug/1000

Chapter 2
Preparing to Use Debug

Chapter 3
Using Debug

Chapter 4
Specifying Locations and Expressions

Chapter 5
Debug Commands

Chapter 6
Profiling

Chapter 7
Xdb Compatibility Mode

Appendix A
Error Messages

List of Illustrations

Tables

1

Introduction to Debug/1000

The HP 92860A Symbolic Debug/1000 (Debug) program is a development tool that allows you to analyze a program and isolate bugs interactively while the program executes. Debug executes on RTE-A and RTE-6/VM based HP 1000 systems. With Debug you can:

Set breakpoints	A breakpoint halts the execution of a program at a particular statement in the program; for example, you can set a breakpoint at a location where a bug is suspected.
Display and modify variables	When Debug is halted at a breakpoint, you can examine or change the value of any variable in your program.
Single-step through a program	Debug can execute one line of code at a time, allowing you to observe the execution of your program statement by statement.
Trace variables	You can instruct Debug to display certain variables every time Debug executes a particular line in your program.
Examine contents of registers	You can display the contents of the A, B, E, O, Q, X, Y, and Z registers.
Show names of calling routines	You can instruct Debug to display the name of the routine currently executing and the names of all routines that called the currently executing routine.
Profile the CPU usage of program modules	You can use the profiling feature of Debug to identify the parts of code using the most CPU time.

Debug is a symbolic debugging tool in that it recognizes symbols used in the program being debugged as well as line numbers. Debug allows you to reference symbols in debugging the program without having to specify the memory addresses for the symbols.

Xdb Compatibility Mode

The Debug/1000 product also provides the Xdb program, an interface to Symbolic Debug/1000 that is similar to the xdb debugger for HP-UX. Xdb executes on RTE-A VC+ systems only. For more information on Xdb compatibility mode, refer to Chapter 7.

Manual Conventions

The following conventions are used in command syntax descriptions:

1. Uppercase characters represent an entry that must be entered exactly as shown.

2. Lowercase italicized characters represent an entry to be determined by the user. If more than one word is used to identify the entry, the phrase is enclosed in angle brackets; for example:

   <*log file*>

3. Optional parameters or operands are enclosed in square brackets. For example:

   V [*number*]

   indicates that the letter V must be entered, optionally followed by a number. If *number* is not specified, Debug uses a default value.

4. Entries that can be repeated one or more times are followed by an ellipsis to indicate repetition. Two examples of the repeated entry are given with the proper delimiter between them. For example:

   [*variable variable*. . .] Specifies an optional variable that can be repeated several times.

   location [*location location*. . .] Specifies that at least one location entry must be given, optionally followed by several location entries.

5. Code and data separation (CDS) information applies only to RTE-A with the VC+ option (RTE-6/VM does not support CDS).

2

Preparing to Use Debug

This chapter describes how to prepare for a debugging session. Compiler and link options that must be used before executing Debug are described in the following sections. How to enter the Debug runstring and the available runstring options are also included in this chapter. Before running a program with Debug, you should be aware of certain Debug limitations described in the following section. Furthermore, Debug must already be loaded in the system. The loading instructions for Debug are provided in the configuration section of this chapter.

Configuration

The HP 92860A product consists of the Symbolic Debug/1000 (Debug) software and this manual, *Symbolic Debug/1000 User's Manual*, part number 92860-90001. The software provided allows you to load and run Debug on an RTE-A or RTE-6/VM Operating System. The User's Manual is a tutorial guide for Debug users which describes all of the Debug commands.

Hardware Requirements

To run, Debug requires an A-Series or E/F-Series HP 1000 computer system. There must be approximately 5900 blocks of disk space on which to restore the software from the media.

Software Requirements

Debug executes on either the RTE-A or the RTE-6/VM Operating System. Normally, the revision of the operating system and Debug must be the most current. Programs to be debugged must be compiled and linked with the symbolic debug option. In the RTE-6/VM environment, the LINK linkage editor must be used to link programs to be debugged.

For RTE-A systems without code and data separation (CDS) or RTE-6/VM systems, Debug requires a partition of approximately 65 pages. For RTE-A systems with CDS, Debug requires a data partition of approximately 95 pages and a code partition of approximately 80 pages. Additionally, another partition is needed in all systems for the program to be debugged. In addition, Debug requires that for each program being debugged at one time, the operating system have one debug table entry and one resource number (RN) made available. Refer to the Memory Allocation Phase chapter in the *RTE-A System Generation and Installation Manual*, part number 92077-90034, or the System and Program Loading Phase section of the *RTE-6/VM Online Generator Reference Manual*, part number 92084-90010.

Debug's VMA size can be defined larger (with the CI or Link VM command), if necessary. The working set size can be increased (with the WS command) to speed up execution if the amount of debug information is large.

When Debug executes, it accesses disk files, creates scratch files, and optionally creates a log file. The scratch files are purged by Debug at the end of the debug session. The amount of disk space required for Debug's scratch file depends on the size of the run file (type 6 file) of the program to be debugged. Currently, this is approximately two times the size of the run file, plus another 32 blocks.

Loading Debug

The following describes how to load Debug:

1. Create a global directory called /DEBUG, which will be your current working directory:

```
CI> crdir /debug
CI> wd /debug
```

2. Restore the Debug software from the product distribution media onto this directory. Refer to either Appendix D in the *RTE-A Primary System Software Installation Manual*, part number 92077-90034, or Appendix E in the *RTE-6/VM Software Installation Manual*, part number 92084-90011, for information on removing files from the media.

3. When the software has been restored onto the directory /DEBUG, install the Debug programs using the CI transfer file INSTALL.CMD. The usage is:

```
[tr,] install [RTEA|CDS|RTE6|XDB] [snap_file] [dest_dir] [NOCAT]
```

where:

`RTEA` loads the RTE-A non-CDS version of Debug.

`CDS` loads the RTE-A VC+ version of Debug.

`RTE6` loads the RTE-6/VM version of Debug.

`XDB` loads the Xdb-compatibility version of Debug (supported on VC+ systems only). This creates the program Xdb.run.

By default, either the RTE-A non-CDS or the RTE-6/VM version of Debug is loaded according to the local system type.

snap_file specifies the snap file for the destination system (default is the local system's snap file).

dest_dir specifies the destination directory for the .RUN file (default is /PROGRAMS).

`NOCAT` inhibits copying the NLS catalogs and CALLS help files to the local /CATALOGS directory.

For RTE-6/VM a copy of Link is made that has fifty pages of EMA to accommodate the required symbol table size.

4. The file DEBUG.SNF is a software numbering file containing a list of the files shipped and their corresponding revision codes. Make a hard copy of this file and save both the file and the copy for future reference.

Xdb may be loaded without including the C language expression-handling code, if desired. This reduces the amount of memory required to execute Xdb by several pages. To remove the C expression handler, modify file XDB.LOD, following the instructions in the comments, and then link Xdb. Only perform this step if you do not require the capability to debug routines written in C.

Loading Errors

The following sections provide two common Debug start-up problems. A complete list of Debug error messages and their explanations are given in Appendix A of this manual.

Chronic Program Abort

If your program aborts constantly with an immediate memory protect upon initialization, you are using a version of the operating system that is not compatible with Debug. The operating system must have the most current date code.

Partition Size Error

If the message

```
Debug: Not enough memory
```

appears, Debug has been loaded in less than 32 pages of memory. Reload it in a 32-page partition and try again.

Debug Limitations

Programs to be debugged must be compiled and linked with the symbolic debug option. Programs or subroutines not compiled and linked with the symbolic debug option cannot be controlled by Debug; therefore, only user-written code can be debugged by Debug. Be aware of the following limitations before running Debug:

1. Normally, only one person can debug a program at a time. To debug a program simultaneously, make another copy of the .DBG file and specify that file with the ‘-d’ Debug runstring option.

2. Some FORTRAN I/O subroutines and system library routines cannot be single-stepped; they must be ‘proceeded’ over. Debug outputs a message to inform you to skip to the next line.

3. Single-stepping over calls to user-written routines that use a non-standard calling sequence may not be possible. Such routines must be proceeded over.
4. Neither EXEC 8 calls nor MLS (Multi-Level Segmentation, produced by MLLDR) programs are supported by Debug. Programs loaded with SEGLD and CDS segmentation are supported.
5. Debug cannot debug any lines of code within a file specified with the INCLUDE directive. Include files with executable code cannot be used with Debug. However, include files with non-executable code in them (for example, data declarations) can be used with Debug.
6. Debug cannot properly display line numbers greater than 32767.

Compiler Options

To use Debug to run a program, you must furnish Debug with the information needed to control program execution. This is accomplished by means of the compiler's symbolic debug option. The symbolic debug option can be included in the source program control statement or appear in the runstring of the compiler or the Macro Assembler. See the appropriate language manual for detailed compiler information.

When the symbolic debug option is specified, the language processor adds symbol table information to the output relocatable file. The symbol table information contains names, types, and locations of all the symbols used in the program, as well as source file names, line numbers, and their locations. Debug later uses this information to build the symbol table.

No extra code is added to a program when it is compiled with the symbolic debug option. This option does not change the way the compiler generates code; it only includes the symbol table information in the relocatable output. Execution of the program is not affected by this option.

LINK Options

Any program to be run with Debug must be linked with the symbolic debug option by LINK. The debug option can be entered interactively as the DE command, in the LINK runstring as the +DE parameter, or in a link command file with the DE command. For example, use the following runstring to link sample program ADDUP with the symbolic debug option:

```
   CI> ru,link,addup.rel,+DE
```

The +DE parameter instructs LINK to relocate the specified file and to prepare the program for debugging.

When the symbolic debug option is given, LINK produces two output files. The first file is the usual type 6 program file. The other file is the debug information file containing line number and symbol table information. If the type 6 file is placed on the CI hierarchical file system directory, the debug information file is given the name of the type 6 file produced, plus a .DBG extension. For example, if the type 6 file is named ADDUP.RUN, the debug information file is named ADDUP.DBG. LINK will overwrite an existing file with that name only if it is a debug information file.

If a type 6 file is placed on an FMGR cartridge, the debug information file is given the name of the type 6 file produced, with an at sign (@) before the first character. For example, if the type 6 file produced is called ADDUP, the debug information file is called @ADDUP. If there is an existing debug information file named @ADDUP, it is overwritten. If there is a file named @ADDUP that is not a debug information file, it is not overwritten, and LINK displays an error message.

If you do not specify the symbolic debug option, LINK will not produce a debug file. If the debug file for a program does not exist, Debug cannot be used to run the program. If an old debug file exists, the information in this file may not be applicable. This can cause misleading information. Therefore, it is recommended that you always specify the LINK debug option when linking a program. This ensures that a current debug file is available for debugging.

The symbolic debug option should be specified even if you do not intend to run your program with Debug. Because Debug does not add any code to your program, the program will run exactly the same way, either with or without the debug option. The only reason not to use the debug option is to reduce the number of LINK output files and the size of the relocatable file.

If a bug shows up in a program which was not linked with the debug option, the program can be relinked with the debug option and then debugged.

Debug Runstring

Once the program is compiled and linked with the symbolic debug option, start the debugging session by running Debug. The two forms of the Debug runstring are as follows:

`[RU,]DEBUG[,-`*option*`,-`*option*`...],`*program*`[/`*session*`]:IH[,`*parm*`,`*parm*`...]`

or

`[RU,]DEBUG[,-`*option*`,-`*option*`...],`*filedesc*`[,`*parm*`,`*parm*`...]`

where:

option — is a Debug runstring option, which always begins with a plus sign (+) or a minus sign (−). See the section on Debug Runstring Options.

program`[/`*session*`]:IH` — a 1..5-character program name, optionally qualified by session number for RTE-A, which is already RPed in the system. This program will be adopted for debugging. Use the −D option when you use :IH, since the location of the .RUN file is not known. For example, "`debug -d mobo.dbg mobo/101:ih`" adopts program MOBO of session 101.

filedesc — the file descriptor of the type 6 file from which Debug clones a new copy of the program and adopts it for debugging. A .run type extension is assumed. For example, "`debug /tent/rentals 5`" clones a new copy of the program in RENTALS.RUN in directory /TENT, passing the runstring parameter 5.

parm — a parameter to the program being debugged.

If there are any errors in the runstring or in the response to the above prompt, Debug reports the error and quits.

For example, entering the following Debug runstring starts a Debug session for the program named TEST:

```
CI> debug test
```

Debug Runstring Options

Several options are defined for use in Debug's runstring. They must appear after the name DEBUG and before the name of the program being debugged, and can appear in any order.

Several options are available:

−B Builds the symbol table, but does not initiate a debug session.

−D Changes the directory containing the information file to the specified directory.

−I Automatic Include command on startup.

−L Redirects the input and output of Debug to another terminal.

−M Causes Debug to immediately enter machine-level mode at the program's primary entry point or point of suspension if active.

−P Schedule with special RMPAR parameters.

−RB Restores the breakpoints that were set when the last debug session exited.

−V Changes the number of source lines displayed in each screen.

−W Inhibits the standard initial schedule, but makes Debug wait for a father program to schedule the program.

The options listed above are explained in the following sections.

−B Option

Purpose: Builds the symbol table, but does not initiate a debug session.

Syntax: `-B`

Description:

Use the −B option to build the symbol table for a program without debugging it. Debug builds the symbol table and terminates after the build is completed. Debug can then run at a later time and will use the symbol table that has already been built for debugging information. An error message is given if the symbol table has already been built (the program has been debugged since the last linking).

Example:

```
CI> debug -b test
```

Builds the symbol table for TEST, but does not begin a debug session.

−D Option

Purpose: Sets the name of the file containing debug information for the program being debugged.

Syntax: `-D` <*file*>

file is the name of the file containing debug information.

Description:

Use the −D option when you use the :IH option because, in this case, the location of the .RUN file is not known.

Example:

```
CI> debug -d ci.dbg ci85a:ih
```

Specifies that the debug information for program CI85A is in file CI.DBG.

−I Option

Purpose: Automatic Include command on startup.

Syntax: `-I,` *cmdfile* `[+]`*logfile*

cmdfile is the name of a file containing one or more Debug commands.

logfile is the log file to which all output from the Debug session is sent.

Description:

The −I option instructs Debug to execute the commands in the command file, and to store all Debug outputs in the log file.

Specifying the plus sign (+) before *logfile* instructs Debug to append to, not overwrite, the log file.

See the section on the Include command in the Debug Commands chapter. Note that the log file entry is required for the −I option.

Example:

```
CI> debug +i my.cmd my.log test
```

Starts a debug session for TEST using the command file MY.CMD and the log file MY.LOG.

−L Option

Purpose: Diverts the input and output of Debug to another terminal.

Syntax: `-L:`*lu*

lu is the terminal logical unit (LU) where the display is to be redirected and where commands to Debug are to be entered.

Description:

The −L option is used for debugging interactive programs that require frequent terminal I/O or utilize many of the special features of the terminal such as graphic displays and block mode reads. Because Debug may interfere with the execution of these programs, the −L option allows you to debug from a terminal other than the one in which your program is running.

All screen displays (including the source lines in the top part of the screen, and all messages and prompts in the bottom part) will appear on the other terminal defined by the *lu* parameter. Commands to Debug also must be entered at this terminal. This option will not affect the program being debugged.

Example:

```
CI> debug -l:32 test
```

Starts a Debug session for TEST, sends the input and output of Debug to terminal LU 32.

−M Option

Purpose: Causes Debug immediately to enter machine-level mode at the program's primary entry point or point of suspension if active.

Syntax: `-M`

Description:

Debug normally operates in source-level mode, which means that source code displays on the screen and the Step command steps source code lines. In machine-level mode, disassembled code displays on the screen and the Step command steps one machine instruction at a time. Entering "`s o`" steps over subroutine calls. The List command displays locations in disassembled format unless a line number is given. All other commands work as normal; it is still possible to use symbols and line numbers as arguments to commands.

Machine-level mode may also be entered by setting the machine level switch "`se ml`". To return to source-level mode, enter "`se ml of`". If ML is ON when Debug performs an initial schedule of a program, Debug does not automatically execute the startup code generated by the compiler, but leaves the program suspended at the primary entry point.

−P Option

Purpose: Schedule the program to be debugged with special RMPAR parameters.

Syntax: `-P:`*n1*`:`*n2*`:`*n3*`:`*n4*`:`*n5*

n1 - *n5*	numeric values to be passed as RMPAR parameters.

Description:

This feature allows you to set the RMPAR parameters to values not derived from the runstring parameter.

The runstring option −P tells Debug to pass the supplied RMPAR parameters directly to the scheduled program, without altering them to the numeric equivalents of the runstring parameters.

When initially scheduled, programs can receive six parameters:

- five 16-bit RMPAR numeric parameters
- one string parameter

Normally, programs are scheduled by an FmpRunProgram call from CI, which sets the RMPAR parameters from the string parameter. Thus, for:

```
ru zippy 12 yow
```

the string parameter is ‘RU,ZIPPY,12,YOW’ and the numeric parameters are:

Numeric Parameter	Value	Meaning
1	12	Numeric equivalent of the first string subparameter
2	YO	First 2 characters of the second subparameter
3	0	
4	0	
5	0	

Debug normally schedules programs in this manner. However, some programs are written to be scheduled by an EXEC call from another program. The EXEC calls allow the numeric parameters to be passed independent of the string parameter. For example,

```
EXEC  (23,5hZIPPY,10,15,20,25,30,6h12,YOW,-6)
```

passes the string parameter “12,YOW”, and the numeric parameters:

Numeric Parameter	Value
1	10
2	15
3	20
4	25
5	30

The −P option allows Debug to schedule programs in this manner. Continuing the above example, to debug program ZIPPY with the same parameters as those passed by the above EXEC call, use:

```
DEBUG -P:10:15:20:25:30 ZIPPY 12 YOW
```

−RB Option

Purpose: Restores the breakpoints that were set when the last debug session exited.

Syntax: `-RB`

Description:

This option restores all of the breakpoints that were set when the last debug session exited, including wakeup counts, command lists, temporaries, and so forth.

−V Option

Purpose: Changes the number of source lines displayed in each screen.

Syntax: `-V:`*nn*

nn must be an integer between 3 and 19, inclusive.

Description:

This option changes the number of source lines displayed in each screen to the specified number. The default number of source lines displayed is 15.

Example:

`CI> debug -v:16 test`	Starts a Debug session and changes the number of source lines displayed in each screen to 16.

See the section on the View command in the Debug Commands chapter.

−W Option

Purpose: Inhibits the standard initial schedule and makes Debug wait for a father program to schedule the program.

Syntax: `-W`

Description:

When Debug is run on a program that has not been initially scheduled, Debug normally performs the initial schedule according to the standard FmpRunProgram rules. The −W option inhibits this schedule, but makes Debug wait for a father program to schedule the program. Once scheduled, the debugging session starts at the first line of code as usual. This allows father programs to set up non-standard string parameters for sons.

Adopting an Existing Program

As shown previously, a program already RPed may be adopted by specifying the program name followed by ":IH", rather than a .RUN file descriptor, in the Debug runstring. Always specify the −D option when adopting an existing program in this manner, so that Debug can find the proper .DBG file for the program.

The −W option, which waits for another program to perform the initial schedule on the adopted program if needed, is often used in conjunction with adopting existing programs. This allows debugging child programs as scheduled by the parent. If the parent schedules the child program with an EXEC schedule call, then the existing program can simply be adopted using the −W option, and the EXEC schedule will be trapped.

If, instead, the parent schedules the child program with an FmpRunProgram call, then the following must take place:

1. The parent's FmpRunProgram schedule parameters must allow scheduling an existing program. That is, the program name portion of the runstring must not contain directory or type extension information.

2. On RTE-6, the child program must be linked with the DC (Don't Clone) command or the FmpRunProgram call will clone a new copy of the program.

3. RP the child program with the proper name.

4. Run Debug on the child program, specifying the −W and the −D options. For example, "`debug -w -d /mondays/child.dbg child:ih`".

5. Now start the parent process. Debug will start up on the first line of the child's code when the parent schedules the child.

Note that if the Debug EX command is entered without the "−R" or "−M" parameter then the child is aborted using a system OF command. This passes back the status value 100000b (−32768) to a waiting parent program, indicating that the child was aborted. If the parent issued an FmpRunProgram call then that call returns FMP error −226, "Program aborted". If you wish the proper status values that the child specifies via a PRTN or PRTM call to be passed back to the parent, then use the EX −R or EX −M command to terminate the Debug session. This allows the child to run to completion, leaving the returned status value intact.

Running Debug

If you have compiled and linked your program with the symbolic debug option, you are ready to run Debug. Run Debug by entering the Debug runstring previously described.

If the symbol table has not yet been built, Debug displays the names of the segments as it builds the symbol table. Debug then displays a block of the source code and prompts for a command with the Debug prompt.

A block of your source code is displayed at the top of the terminal screen during the Debug session. The display is adjusted as the Debug commands are executed. Debug shows where the program is halted by a current line marker >. At the beginning of the Debug session, the current line marker is at the first executable line of code. All subsequent source code listings are adjusted so that the current line marker is positioned at the center of the block. The bottom half of the terminal screen is used to enter Debug commands.

As Debug executes, a screenful of source code is displayed. The display is memory locked if the terminal has that capability. For terminals that do not have the capability or for an HP 2621 terminal, Debug simulates memory lock by maintaining a record of the lines displayed and scrolling only the Debug display and operator entries, leaving the source code undisturbed.

You can enter any of the Debug commands after the Debug prompt, Debug>.

When Debug is in the single-stepping mode, the prompt changes to:

```
Step>>  _
```

When Debug is in the listing mode, the prompt changes to:

```
List>>  _
```

When Debug is in the profiling mode, the prompt changes to:

```
Profile>  _
```

At the Debug prompt, the program is under Debug control and execution will not occur until Debug allows it to do so. The general format for entering instructions to Debug is:

command operand1 operand2 . . . operandn

where:

command — is a Debug command. Refer to the appropriate command description for acceptable abbreviations for individual commands.

operand1 through *operandn* — are the legal operands for the command entered. Refer to the chapter on Debug Commands for a description of the operands allowed for each command.

Debug accepts lowercase letters for commands and operands. Debug commands and operands are separated by a space, not a comma; multiple commands on a line may be entered when separated by a semicolon (;). When Debug encounters a semicolon, it assumes that the line contains multiple commands.

If you want to include a semicolon (;) as part of a command argument, enter two semicolons in a row. The two semicolons will be interpreted as a single semicolon to be included in the argument. For example, the command "`m jetski 'wet;;wild'`" will modify variable `jetski` to the value `wet;wild`.

The default conditions associated with Debug commands are contained in the command descriptions. However, Debug displays messages under certain situations such as an illegal entry and requests a yes/no answer. The default entry is given in the display in square brackets. Press the return key to specify the default entry.

Basic Commands

A short summary with examples of the most common commands follows. Complete step-by-step examples can be found in the Debug Commands chapter.

Break Command

The Break command sets a breakpoint at the line specified. A breakpoint causes Debug to halt execution when it reaches a particular statement in a program. Breakpoints can be specified within a different routine than the currently executing one by appending a slash and the routine name to the line number. Breakpoints can also be set to break after a certain number of iterations through the breakpoint (as in a loop). Condition breakpoints break contingent on the value of a variable.

`Debug> b 20`	Sets a breakpoint at line 20 in the currently executing routine.
`Debug> b 20/sub1`	Sets a breakpoint at line 20 in subroutine SUB1.
`Debug> b 20 sum > 50`	Sets a conditional breakpoint. Instructs Debug to break at line 20 in the currently executing routine only if the value of SUM is greater than 50.

Proceed Command

The Proceed command instructs Debug to proceed to a breakpoint or specified line number. If no line number is specified, the program runs to the next breakpoint or to completion if no breakpoint is encountered.

`Debug> p 30`	Proceeds to line 30 in currently executing routine.
`Debug> p`	Proceeds to next breakpoint or to completion if no breakpoint is encountered.

Display Command

The display command displays the value of one or more variables.

`Debug> d sum`	Displays the current value of variable SUM.
`Debug> d x y z`	Displays the current value of variables X, Y, and Z.

Modify Command

The Modify command modifies the value of a variable.

`Debug> m sum 100`	Modify the value of SUM from its current value to 100.

Debug will display a message showing the new value of the variable.

`SUM: 50 => 100`	The value of SUM was modified from 50 to 100.

Help Command

The Help command (?) displays a short summary of the Debug commands available. Enter the Help command followed by a command (? command); a brief description of that command is displayed.

Exit Command

The Exit command terminates your program so that your program will not continue to completion. You can use this command at any time to end the Debug session.

`Debug> ex`	Exits Debug.

3

Using Debug

This chapter is a tutorial guide for the novice Debug user. It gives information such as starting a Debug session, using breakpoints, displaying and modifying variables, and stepping through the program. Step-by-step examples of using Debug are also included.

Sample Source Code for Debug Examples

The following sections contain examples illustrating some of the Debug commands. The various steps in the debugging process also are described. A sample program follows and is used to demonstrate the Break, Display, Clear, Step, Modify, Where, Find, and Exit commands. This program opens a file named NUMBER, which resides on global directory JOE, reads a series of numbers from the file, totals them, and prints the total.

Note that this example is a FORTRAN program and must be compiled using the FTN7X “s” compiler option given in the compiler runstring to produce symbolic debug information; for example, “`ftn7x addup.ftn 0 -,,s`”. This program must then be linked using the link debug option, for example, “`link addup.rel +de`”.

```
$files 0,1
c
         program addup(4,99),try out symbolic debug
         implicit none

         integer*2 FinalTotal,ios,number

c        This program reads a set of numbers from
c        a file and adds them up.

         call init

         FinalTotal = 0
         do while (.true.)
            read(100,*,end=99,iostat=ios) number
            if (ios.ne.0) then
                 write(1,*) ’Error #’,ios
                 stop
            else
                  FinalTotal = FinalTotal + number
            endif
         end do
```

```

99      write(1,*) 'The final total is:  ',FinalTotal

        call cleanup

        end


        subroutine init
        implicit none

        integer*2 ios
        character*6 dir

        dir = 'JOE'

        open(100,file='NUMBER::'//dir,iostat=ios)
        if (ios.ne.0) then
                write(1,*)'Can''t open, error #',ios
                stop
        endif

        return
        end


        subroutine cleanup
        implicit none

        close(100)

        return
        end
```

In the example, the file NUMBER exists on global directory JOE and contains the following ASCII numbers:

```
12

34

99
```

Starting a Debug Session

To start the debug session after the program has been compiled and linked with the symbolic debug option, enter the Debug runstring:

[RU,]DEBUG[,–*option*,–*option*...],*program*[/*session*]:IH[,*parm*,*parm*...]

or

[RU,]DEBUG[,–*option*,–*option*...],*filedesc*[,*parm*,*parm*...]

Refer to Chapter 2 for runstring parameter definitions.

When the Debug runstring is entered, Debug prepares the program for execution and builds the symbol table. The source file is displayed at the top half of your terminal screen and the Debug commands can be entered in the bottom half. At the beginning of a debug session, the current line is the first executable line of code. Using the example source code shown, your screen should display something similar to the following:

```
 04          implicit none
 05
 06          integer*2  FinalTotal,ios,number
 07
 08 c        This program reads a set of numbers from a file and
 09 c        adds them up.
 10
>11          call init
 12
 13          FinalTotal = 0
 14          do while (.true.)
 15          read(100,*,end=99,iostat=ios)  number
 16          if (ios,ne.0) then
 17                  write(1,*) ’Error #’,ios
 18                  stop
 File:       /TEST/ADDUP.FTN
 Debug>
```

As indicated above, Debug displays the name of the file on-screen between the source window and the command area. It appears in an inverse video banner on terminals with that capability, or underlined on others.

The “Debug>” prompt indicates that Debug is ready to accept the next Debug command. Note that in the program listing, the arrow always points to the line of source code to be executed. Debug positions the source file so that the arrow is always near the middle of the lines displayed.

In addition to the “>” line marker, Debug also uses a tilde (~) to indicate approximately the next line when the current position is somewhere inside a line. Debug uses an equals sign (=) to mark the appropriate line containing strings found with the Find command.

Using Breakpoints

The following sections describe how to set, clear, and display breakpoints and conditional breakpoints. Each breakpoint has a number associated with it at the time it is set. This number is displayed whenever the breakpoints are listed with the ‘B’ command, for example:

```
Debug> b
Breakpoints:
# 1 at 78/ZIPPY
# 2 at 80/ZIPPY
  if:  cabbage = absent
# 3 at 80/ZIPPY
   trace:  bobcat\4
```

This number is used to refer to the individual breakpoint by commands that accept breakpoint numbers as arguments. Clear breakpoint (CB), Activate breakpoint (AB), and Deactivate breakpoint (DB) are commands which can accept references to individual breakpoints.

Setting a Breakpoint and Proceeding

The Debug commands for setting and proceeding to a breakpoint are B and P respectively. The following example shows setting a breakpoint at line 13 and proceeding to that breakpoint. Debug executes the program and halts execution at line 13. The top half of the terminal will scroll up and the arrow will point to line 13. Your screen should be similar to the display shown below:

```
 06
 07
 08 c        This program reads a set of numbers from a file
 09 c        and adds them up.
 10
 11          call init
 12
>13          FinalTotal = 0
 14          do while (.true.)
 15          read(100,*,end=99,iostat=ios)  number
 16          if (ios.ne.0) then
 17                  write(1,*) ’Error #’,ios
 18                  stop
 19          else
 20          FinalTotal = FinalTotal + number

 Debug> b 13
 Set breakpoint # 1 at 13/ADDUP
 Debug> p
 Break at 13/ADDUP
 Debug> _
```

At this point in the execution of the program, subroutine INIT has been called. The arrow is placed on line 13 to indicate that this is the next line to be executed.

Setting Conditional Breakpoints

A conditional breakpoint is a breakpoint that has an associated variable and a condition to test that variable. Debug tests the variable and allows the breakpoint to take effect only if the condition is true. (Actually, the program stops each time it reaches that line, but Debug restarts the program if the condition is not true.)

The following example sets a conditional breakpoint at line 20. In this example, Debug stops program execution at line 20 only if the variable FINALTOTAL exceeds 34. The screen display when the program halts at line 20 is shown:

```
 13          FinalTotal = 0
 14          do while (.true.)
 15          read(100,*,end=99,iostat=ios)  number
 16           if (ios.ne.0) then
 17                 write(1,*) 'Error #',ios
 18                 stop
 19          else
>20          FinalTotal = FinalTotal + number
 21          endif
 22          end do
 23
 24 99       write(1,*) 'The final total is:  ',FinalTotal
 25
 26          call cleanup
 27
```

```
Debug> b 20 FinalTotal > 34      Conditional breakpoint set at line 20.
Set breakpoint # 2 at 20/ADDUP
Debug> p                         Starts program execution.
At 20/ADDUP FINALTOTAL > 34      Debug halted the program at line 20.
Debug> _
```

Note that the DO loop in program ADDUP has been executed twice before the value of FINALTOTAL exceeds 34.

The conditional operators for setting conditional breakpoints are:

=	equal
< >	not equal
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Debug does a substantial amount of processing each time the program reaches a location with a conditional breakpoint attached, whether or not the condition is true. Therefore, if you use conditional breakpoints at locations that are executed many times, execution may be slow. Refer to the Break command description in Chapter 4 for more information on conditional breakpoints.

Displaying and Clearing Breakpoints

To display all breakpoints including conditional breakpoints and tracepoints, enter the Break command without an operand.

```
Debug> b
Breakpoints:
  # 1 at 13/ADDUP
  # 2 at 20/ADDUP FINALTOTAL > 34
Debug> _
```

The list of breakpoints are displayed in the order of entry. Each line number is displayed with the program or subroutine where the line resides.

The Clear (C) command clears a previously set breakpoint. You must specify the location where the breakpoint was set. Using the sample program ADDUP, to clear the breakpoint set at line 13, enter:

```
Debug> c 13
Cleared breakpoint # 1 at 13/ADDUP
Debug> _
```

To clear all the breakpoints previously set, enter the clear command followed by an asterisk:

```
Debug> c *
All clear
Debug> _
```

Displaying and Modifying Contents of a Location

You can display or modify the contents of locations any time Debug prompts for a command. The next two sections show how to use the Display and Modify Commands.

Using the Display Command

The contents of locations can be examined using the Display command. After setting a conditional breakpoint and executing the program, display the contents of FINALTOTAL:

```
Debug> d FinalTotal
FINALTOTAL = 46
Debug> _
```

More than one location can be specified. Debug displays the value of each variable requested and prompts for another Debug command.

If you enter the display command with no operands, Debug displays the variable or variables specified in the previous display command. For example:

```
Debug> d
FINALTOTAL = 46
Debug> _
```

You can display a variable in a different format by appending an override operator to the variable. Refer to the Override Operators section in chapter 4 for more details.

Using the Modify Command

The Modify (M) command changes the value of a variable in the program. For example, to change the variable FINALTOTAL in sample program ADDUP to 1234 and display the new value:

```
Debug> m FinalTotal 1234
FINALTOTAL:   46 => 1234
Debug> _
```

The value of FINALTOTAL is changed from 46 to 1234.

```
Debug> d FinalTotal
FINALTOTAL = 1234
Debug> _
```

Display the value of FINALTOTAL.

Exiting Debug

To exit Debug, use the Exit command. The Exit command can be specified either as E, which asks if you really want to exit, or as EX, which will not ask. Examples:

```
Debug> e
Really quit? [y]
CI> _

Debug> ex
CI> _
```

Pressing the return key in response to the question is taken as a “yes” answer. If the Exit command is given from Profiling mode, you are always asked if the overview file should be purged.

Entering EX or E and “yes” to the prompt terminates your program. Your program will not execute to completion. You may use this command any time you are through with the debugging session. Upon normal termination, control is returned to the program that scheduled Debug, usually CI or FMGR.

If you enter E and “no” to the prompt, Debug prompts for another command.

Using the Step Command

The Step (S) command is used to step from one line of executable source code to the next. After the S command is entered, Debug interprets all subsequent carriage returns as step commands until another Debug command is entered. To remind you that carriage return is interpreted as a step command, Debug changes the prompt to “Step>.”

Only executable lines can be single-stepped. While in the single-step mode, lines that do not produce executable code are skipped by Debug.

Some system library routines may contain code not compiled with the symbolic debug option and are too complicated for Debug to single-step quickly. When Debug cannot single-step, a message similar to the one below is displayed:

```
Stopping single step:  too many instructions.
Proceed to 16/ADDUP? [Y] _
```

The above message displays because line 15 of program ADDUP calls a system routine that takes a long time to single-step. Debug expects a yes or no answer after displaying the above message. Either a “Y” or carriage return causes Debug to set a temporary breakpoint at line 16 and then execute the program up to line 16. If line 15 does a branch to a line other than 16 or if you are not sure that line 16 is the next line to be executed, you must answer “no” to the above prompt. Entering “N” causes Debug to prompt:

```
Continue stepping? [Y] _
```

If you enter “Y”, Debug starts again and continues the single-step operation until it is done or until too many instructions go by again. If too many instructions go by again, the process is repeated.

If you enter “N”, Debug leaves the program suspended inside the library routine.

A “Proceed to ?” prompt also may occur when Debug encounters a reference to an unknown instruction. These instructions may be privileged, illegal, or customized instructions whose destinations are unknown to Debug.

```
Can’t single step: unknown instruction.
Proceed to 2817/Elsinore? [Y] _
```

Breakpoints must be placed at all the possible destinations of the line that could not be single-stepped and the Proceed command given. If you are not sure where all the possible destinations are, you can Proceed to a location to which you know control will eventually return. Note that Debug may have partially executed the line, and your program may be suspended at a module without debug information. If so, qualify all locations with the desired module name. Also, give the desired segment name if the same routine exits in more than one segment.

Exit the Debug session with the Exit command and start another Debug session with the sample program ADDUP. Proceed to line 15, and when Debug is halted at line 15, enter the Step (S) command. As the program is being single-stepped, the source code display is scrolled up with the line marker pointing to the next line to be executed. Assuming the sample program is halted at line 15, the complete sequence of the single-step mode is shown below:

```
Debug> s
Stopping single step:  Too many instructions.
Proceed to 16/ADDUP? [Y] y
Step>>                         Debug moves to line 16.
Step>>                         Debug moves to line 20.
```

or

```
Proceed to 16/ADDUP? [Y] n
Continue stepping? [Y] y
Step>>                         Debug moves to line 20.
```

At this point, the Y answer caused program execution to proceed to line 20, indicated by the current line marker at that line. The lines skipped are the ones that are not executed because the IF condition is false. To continue single-stepping:

```
Step>>                         Debug moved to line 22.  Line 21 is skipped (no
                               executable code).
Step>>                         Debug moved to line 15.
Step>>
Stopping single step:  Too many instructions.
```

Stepping Over a Subroutine

Exit the debug session with the Exit command and start another session with the sample program ADDUP. After starting the debug session, Debug stops the program at the first executable line (line 11). To step through a subroutine, the program being debugged must be halted at the line that calls the subroutine.

```
 04          implicit none
 05
 06          integer*2  FinalTotal,ios,number
 07
 08 c        This program reads a set of numbers from a
 09 c        file and adds them up.
 10
>11          call init
 12
 13          FinalTotal = 0
 14          do while (.true.)
 15          read(100,*,end=99,iostat=ios)  number
 16          if (ios.ne.0) then
 17          write(1,*) ’Error # ’,ios
 18          stop

 Debug> _
```

Line 11 calls subroutine INIT. Enter the S command and Debug steps through the subroutine, returning to line 13. At this point, all the lines in INIT have been executed. The screen display and the Debug dialog are shown below.

```
 06          integer*2  FinalTotal,ios,number
 07
 08 c        This program reads a set of numbers from a file
 09 c        and adds them up.
 10
 11          call init
 12
>13          FinalTotal = 0
 14          do while (.true.)
 15          read(100,*,end=99,iostat=ios)  number
 16          if (ios.ne.0) then
 17                  write(1,*) 'Error # ',ios
 18                  stop

 Debug> s
 Debug> _
```

Stepping Into a Subroutine

Exit the debug session with the Exit command and start another session with the sample program ADDUP. Enter the Step Into (S I) command with the program halted at line 11. The source code displayed will change to the following:

```
 30
 31          subroutine init
 32          implicit none
 33
 34          integer*2 ios
 35          character*6 dir
 36
>37          dir = 'JOE'
 38
 39          open(100,file='NUMBER::'//dir,iostat=ios)
 40          if (ios.ne.0) then
 41                  write(1,*) 'Can''t open, error #',ios
 42                  stop
 43          endif
 44

 Debug> s i
 At 37/INIT
 Step>> _
```

After the Step Into command is entered, the current line marker points to the first executable line of subroutine INIT which is line 37. You can single-step through this subroutine. When you step through the RETURN statement, Debug returns to the main program, at line 13.

```
 06          integer*2  FinalTotal,ios,number
 07
 08 c        This program reads a set of numbers from a
 09 c        file and adds them up.
 10
 11          call init
 12
>13          FinalTotal = 0
 14          do while (.true.)
 15          read(100,*,end=99,iostat=ios)  number
 16          if (ios.ne.0) then
 17                  write(1,*) 'Error # ',ios
 18                  stop
 19          else
 20          FinalTotal = FinalTotal + number
```

```
Step>>
At 13/ADDUP
Step>> _
```

Stepping through the RETURN statement returns Debug to the main program at line 13.

You can continue debugging the sample program (halted at line 13 of the main) by entering any Debug command.

Using the Where Command

To find out the current subroutine that Debug is executing and the names of the routines that called it, use the Where (W) command. If any parameters were passed, they are displayed also. For example, after stepping into the subroutine INIT, enter the where command:

```
Debug> w
At 39/INIT
Callers:
   11/ADDUP
Debug> _
```

Accessing Another Subroutine

A subroutine other than the currently executing one can be accessed by appending a slash and the subroutine name in a location parameter. For example, to display the variable DIR in the subroutine INIT:

```
Debug> d dir/init
DIR/INIT = 'JOE'
Debug> _
```

In the sample program ADDUP, to set a breakpoint at the first executable line of subroutine CLEANUP, enter the following:

```
Debug> b cleanup
Set breakpoint # 1 at 52/CLEANUP

Debug> _
```

A breakpoint is set at the first executable line in subroutine CLEANUP (line 52 of the source).

To set a breakpoint at a particular executable line in a subroutine, enter the line number, a slash, and the subroutine name. For example, to set a breakpoint at line 39 of subroutine INIT, enter:

```
Debug> b 39/Init
Breakpoint set at 39/INIT
Debug> _
```

A breakpoint is set at line 39 in subroutine INIT.

4

Specifying Locations and Expressions

Specifying Locations

A location is a Debug command operand that specifies an address for the given command. Debug accepts names of routines, variable names, register names, statement labels, source file line numbers, and absolute addresses as locations.

To specify a location, use the following syntax:

ref[*subscr*] [*selector*] [*qual*]

where:

ref specifies the actual reference type (the base on which the other fields operate). Possible values are as follows:

register specifies a register (A, B, E, O, Q, P, or Z) and must be surrounded by parentheses or square brackets. For example, "`d [a]`" displays the A-Register.

line# is a source file line number, possibly ending in 'l'. For example, "`b 42`" sets a breakpoint at line 42 of the current routine, and "`d 42l`" displays the contents of line 42.

address is a numeric address, ending in 'a', 'b', 'c', 'e', or 'q'. For example, "`d 2551c`" displays the contents of octal address 2551 in the code partition.

symbol is a user-defined symbol, either a variable name, statement label, or entry point name. For example, "`b #700`" sets a breakpoint at statement lable 700, and "`p cabbage`" proceeds to routine CABBAGE.

label is a pound sign (#) followed by a FORTRAN statement label number or a C statement label identifier. Examples for FORTRAN and C are as follows:

FORTRAN:	`b #700`	at label 700
C expression:	`b #cake`	at label cake

constant is a 16-bit octal constant ending in 'o', or a 32-bit decimal constant. 32-bit constants are only possible if numeric constants are not line numbers (see Expressions later in this chapter).

. denotes the current location. For example, "`t .`" sets a tracepoint at the current location.

subscr specifies subscripts into an array. Subscripts may be given as comma-separated values enclosed in parentheses or square brackets. Each subscript may be enclosed in square brackets. For example:

```
both      d coleslaw[mayo,2]       and
          d coleslaw[mayo][2]
```

display the value of element `[mayo,2]` in the two-dimensional array `coleslaw`.

selector specifies a particular component of a structured variable. The selector must conform to the syntax:

{^|.*subfld*}[*subscr*][*selector*]

where:

^ is the Pascal pointer de-referencing character.

subfld is the name of a Pascal record subfield or C structure/union member. For example,

```
d canal.water[q^.fullstack[2]]
```

qual this qualifies the *ref* portion of the location with the name of the routine that contains that location. For example, the qualifier might specify the name of the routine in which a variable is defined. This is necessary if the containing routine is not the currently-executing routine. The qualifier syntax is:

```
         {         /=       }
[/line#] { [/routine/[routine]] } [/filename] [/segment] [/depth]
         {         /.       }
```

where:

line# is an executable line number within the routine given by the rest of the qualifier. This is useful for qualifying variables which reside in C unnamed blocks. For example, "`adam/12/im`" specifies variable `adam` that is defined within the scope of line 12 of routine `im`.

= specifies the currently-listed location. For example, "`more/=`" specifies variable `more` within the routine on the display.

routine is the name of a routine, such as a FORTRAN subprogram, C function, or Macro entry point. Multiple routine names specify Pascal nested procedures.

filename specifies the file name for the preceding reference, in filename.ext syntax without directory path or other file descriptor information. This is useful for qualifying routine names which are duplicated within the program, such as C static routines. For example, "`cling/free.c`" specifies static routine `cling` which is defined within file `free.c` . C file-level global variables may also be specified in this manner; "`warming/bench.c`" specifies global variable `warming` which is defined in variable `bench.c` .

segment	specifies a particular overlay (SEGLD) segment name, or a string of form SEG.*nn*, where *nn* is the CDS segment number.
depth	specifies the CDS stack depth of the desired routine invocation. This is useful for displaying CDS stack-relative locals and parameters for other than the most recent invocation of the specified routine. For example, “`cornnuts/jello/2`” specifies local variable `cornnuts` contained in routine `jello` at CDS stack depth 2.

Note that the above syntax prohibits the use of symbols in Pascal programs that contain the characters ‘`.`’ and ‘`^`’. To specify a symbol that contains an invalid character, surround the symbol with back quotes (‘) as in ‘PAS.STOP‘/seg1. Characters entered inside back quotes are not automatically converted to uppercase; therefore, entries for all symbols entered between back quotes must be in uppercase. Also, refer to the SET command in the Debug Commands chapter.

When Debug prints a message about a location, it uses the following format:

`line/module[/segment][offset]`	Location known to Debug.
or	
`octal  address/??[/segment]`	Location unknown to Debug.

The offset is a positive or negative octal number of words giving the displacement of the location from the line shown. The offset is printed only if it is non-zero.

Scoping Rules

The scope of a variable is the routine or routines for which the variable is defined. When Debug is given a symbol as part of a location, it must search its symbol table to find the address assigned to that symbol by the compilation and linking process. Each routine has a particular set of symbols defined for that routine. Therefore, a reference to a symbol in one routine may or may not access the same symbol in another routine.

You can specify variables as either qualified or unqualified symbols. Qualified symbols tell Debug the exact routine to find the variable in. Unqualified symbols are variables specified without any information about where to find that variable. As an example, assume that a variable named ZIPPY exists in both subroutines SUB1 and SUB2. If you are currently in subroutine SUB1, and want to display the variable ZIPPY in subroutine SUB2, and you enter:

`Debug> d zippy`	Debug displays the value of ZIPPY in the current subroutine, SUB1.

To specify the desired variable without ambiguity, enter:

`Debug> d zippy/sub2`	Debug displays the value of ZIPPY in subroutine SUB2.

Debug uses the following search sequence when trying to locate unqualified symbols:

1. If the symbol is defined to the currently executing routine, that definition is used.

2. If the symbol is defined in an outer-level routine which the current routine is nested within, including the main program, that value is used. This applies to Pascal routines only.

3. If the symbol still is not found, Debug gives the error message “Symbol not found” and does not execute the command.

An example of an unqualified symbol:

`Debug> d sum`	Debug follows the above scoping rules to find the specified variable.

Qualified symbols tell Debug the exact routine in which to find the variable. To reference a symbol which is defined to a routine other than the currently executing one, the location must be qualified by the desired routine. Symbols are qualified by appending a slash (/) and the routine name of the location. For example:

`Debug> d q/subx`	Displays variable Q of subroutine SUBX.
`Debug> b #10/init`	Sets a breakpoint at FORTRAN statement 10 in subroutine INIT.

Pascal programs can have nested routines, complete routines that are local to other routines. These routines can be qualified by entering nested procedure and function names between slashes. For example:

`Debug> b city/county/state`	Sets a breakpoint at routine CITY, which is nested within routine COUNTY, which is nested within routine STATE.

Segment names can also be used to fully qualify the location. This is used only for SEGLD segmentation, where a routine with the same name can appear in more than one segment. If a routine in a segment is not qualified, Debug first searches for it in the currently loaded segment. If the routine is not found, Debug searches all the segments in the order that LINK relocated them. If the routine is still not found, Debug displays an error message. For example:

`Debug> b city/seg1`	Sets a breakpoint in routine CITY in segment SEG1.

Note that since a routine can appear in only one CDS segment, it is not necessary to qualify routines by segments when debugging CDS programs.

The same scoping rules do not apply to line numbers. Only the line numbers for the currently executing routine can be unqualified. All others must be qualified by subroutine name. For example:

`Debug> L 200`	Lists line 200 of currently executing routine.
`Debug> L 200/sub3`	Lists line 200 of subroutine SUB3.

Function Return Variables

Within the body of a function, the name of the function refers to the return value as in FORTRAN or Pascal. Outside the function, its name refers to the entry point by which the function is called. For example, within function BLENDER, “d blender” displays the current return value. Outside BLENDER, it displays the value of the entry point.

Arrays

Array subscripts may be entered as comma-separated values enclosed in parentheses or square brackets, or each subscript may be enclosed in square brackets. For example:

```
Debug> d xyz(3)
XYZ(3) = -2660
```

Displays the third element of the array XYZ.

If a location was not declared as an array in the source code, Debug can still use a single subscript to act as an offset from the address of the location. In this case, Debug assumes the location is a variable with a lower bound of 0 for C or 1 for all other languages, with the size of each element being the size of the variable itself. For example:

```
Debug> d abc:i4
ABC:I4 = 906 609 -906 -609
```

Displays ABC and the next three values.

```
Debug> d abc[3]
ABC[3] = -906
```

Displays the ‘third element’ in the ‘array’ ABC.

Up to 7 dimensions are supported, with extra subscripts ignored, and missing subscripts filled in with their lower bounds. For example:

```
Debug> d i xyz[i,1]:o
I = 2  XYZ[I,1]:O = 0b
```

Displays variable I and the first element in row 2 of array XYZ.

```
 Debug> d table[1,5,66]
TABLE[1,5,66] = 33
```

Displays the selected element of TABLE.

```
Debug> d table
TABLE = 69
```

Displays the first element of TABLE.

```
 Debug> d a[i,j,k,last]
A[I,J,K,LAST] = 9999
```

Displays the value residing at the appropriate location in A.

```
d a[i][j][k][last]
a[i][j][k][last] = 9999
```

Displays the same element using C syntax.

```
Debug> _
```

Subscripts can be any expression. If not enough subscripts are provided, Debug uses the lower bound of those dimensions not given. If too many subscripts are entered, Debug ignores those not needed. If the subscripts entered result in a final value residing outside the array, Debug still displays that value, even though it may be unintelligible data.

Pascal Packed Arrays of Character

You do not need to specify either a type override directive or a repeat count for objects declared as a packed array of CHAR in a Pascal program. A variable of this type might be declared like this:

```
VAR  pac10:  PACKED ARRAY [1..10] OF CHAR;
```

According to the type override rules, you would usually enter:

```
Debug> d pac10:c10
```

to display this variable. However, Debug knows all about the array, so the name of the variable is the only item that needs to be entered. Debug will fill in the rest, and display the 10 characters automatically. The default type/format for Pascal Packed Arrays of CHAR is a string of characters as long as the subscript range for the array.

```
Debug> d pac10
PAC10 = 'HELLO JANE'
Debug> _
```

BASIC Array Limitations

When a BASIC array is declared using the DIMENSION statement, the compiler does not emit any debug information for it. Therefore, Debug will not be able to display or modify such objects.

However, when a variable is declared as an integer having a number of elements, for example:

```
INTEGER I(0:5),J(-906:50)
```

debug information is emitted. Debug knows about it as a simple variable, however, not as an array. This means that Debug is not aware that the array J is indexed from −906 to 50, requiring you to calculate the index. Debug assumes the lower bound of your array starts at one. To calculate the correct upper bound for Debug, use the following formula:

```
upper bound = (upper limit - lower limit) + 1
```

For example:

```
Debug> d i[1]                Displays the first element of the array I.
I[1] = 5                     Debug thinks the indexes start at 1, not 0.

Debug> d j[957]              Displays the last element of the array J.
J[957] = 33                  This is the same as j(50) in the BASIC program.
Debug> _
```

Using Expressions

Most commands that accept locations also accept expressions as operands. To use an expression, use the following syntax:

subexpr [*format*]

where *subexpr* can be expressed as any of the following:

location
unaryop subexpr
subexpr binaryop subexpr
subexpr postop
(subexpr)
subexpr :type
subexpr<substr>
&dsv

Spaces are included in the above syntax for purposes of legibility only. They are not part of the syntax and should not be entered.

location is any location.

unaryop is one of the following unary operators: *, −, \, @, ’, |NOT|, |BITS|, |BIT|, |LEN|, |MAXL|, |FSTR|, or |LABL|.

binaryop is one of the following binary operators: +, −, *, /, **, |AND|, |OR|, |XOR|, |SHFT|, |IROT|, |JROT|, |DIV|, or |MOD|.

postop is one of the following postfix operators:

.member
−>member
[subscr]
<substr>
:type

dsv is a Debug Session Variable, see Debug Commands, Chapter 5.

where *format* is expressed as:

[*displaytype*] [*repeat*]

where:

displaytype is the type to use when displaying the object, but does not change the ‘type’ discussed below.

repeat is a numeric constant between 1 and 32767 or another expression enclosed in square brackets, specifying how many contiguous objects are to be displayed.

Operators

Operators are of four types: arithmetic, attribute, address, and override. Multiple operators on the same line are evaluated according to their level of precedence. Additional topics related to operators are also discussed in this section.

Arithmetic Operators

Arithmetic operators expect operands of 1- or 2-word integer type, and return values of 2-word integer type. The '/' can be used as an operator, but it also denotes a qualifier in the location syntax. For this reason, surround the first operand with parentheses if any confusion could exist or use the |DIV| operator, which does the same thing. The arithmetic operators are:

Expression	Description
i+j	I plus J
i−j	I minus J
i*j	I times J
i|DIV|j	I divided by J
i**j	I raised to the power J
−i	negative of I
i|MOD|j	I modulo J, remainder of integer division
i|AND|j	I ANDed with J
i|OR|j	I ORed with J
i|XOR|j	I exclusive-ORed with J
i|SHFT|j	I logical-shifted J bits: shift left if J>0, shift right if J<0
i|IROT|j	lower 16 bits of I rotated J bits, left/right as above
i|JROT|j	32-bit rotate of I, as above
|NOT|i	boolean NOT on I

Attribute Operators

Attribute operators accept any location and return a 32-bit integer value describing the characteristics of that location. The Attribute operators are:

Expression	Description
\a	the word address of A
|BITS|a	the number of bits contained by A
|BIT|a	the bit offset within the first word where A starts
|LEN|a	the current length of string A
|MAXL|a	the maximum length of string A

Address Operators

Address operators accept any location and return a modified address within the program. This modified address is the location within the program that will be displayed if the result is evaluated. The Address operators are:

Expression — Description

@a — Returns the contents of the first word of A with indirection resolved as a program address. For example, “`d @zarquan`” displays the contents of the location pointed to by ZARQUAN.

’a — Returns the contents of the first word of A as the byte address of a program location. For example, “`d ’zarquan`” displays the byte pointed to by ZARQUAN.

*a — Returns the object pointed to by C pointer “a”. For example, “`d *whoa`” displays the object pointed to by `whoa`.

a.m — Returns member “m” of C structure or union “a”. For example, “`d (*whoa).naugahyde`” displays member `naugahyde` of the structure pointed to by `whoa`.

a–>m — Returns member “m” of the C structure or union pointed to a “a”. For example, “`d whoa–>naugahyde`” displays member `naugahyde` of the structure pointed to by `whoa`.

a[subscr] — Returns the specified array element. Subscripts may be separated by commas and enclosed in a single pair of brackets or parentheses. Alternately, each subscript may be entered as separate values. For example, both “`d bazooka[limbo, mania]`” and “`d bazooka[limbo][mania]`” display element `[limbo,mania]` of array `bazooka`.

|FSTR|a — Given the address of a FORTRAN string descriptor, returns the program location of the first byte of the string, automatically typed as a string of the length given by the descriptor. For example, “`d |fstr|2314b`” displays the contents of the string whose descriptor is at octal address 2314b.

|LABL|a — Returns the program address corresponding to CDS Label A. For example, “`b |labl|2552c` ” sets a breakpoint at the entry point given by the CDS Label in address 2552 octal in the code partition.

Override Operators

Override operators take a subexpression and change its 'type,' that is, the size of the object. Type overrides are discussed later in this chapter. The Override operators are:

<u>Expression</u> <u>Description</u>

a:type — Changes the size of "a" to the number of words in an object of type type. The type values are listed later in this chapter.

a\format — Changes the display format of "a" to the object type and repeat count specified. Note that this must appear at the end of the expression.

<substr> — Selects specific characters of a string defined in terms of character positions. For example, if `str = 'copyright'`, then

`str<3:5>`	displays the string	`pyr`
`str<3>`	displays the string	`p`
`str<3:>`	displays the string	`pyright`

a<3:7> — Changes the size of "a" to 5 bytes, starting at byte (character) 3.

Operator Precedence

Multiple operators on the same line are evaluated according to their level of precedence, with the exception of operators contained within parentheses. The Operator Precedence is:

Precedence Level	Operators
0	\format
1	|AND|, |OR|, |XOR|, |SHFT|, |IROT|, and |JROT|
2	+, −, |NOT|
3	a*b, /, |DIV|, and |MOD|
4	**
5	:type <substr>
6	*,\, @, ', |BITS|, |BIT|, |LEN|, and |MAXL|
7	.member, −>member, [subscr]

The greater the precedence level number, the higher their priority is over other operators. That is, if an operator has a precedence level of 2, and another operator has a precedence level of 3, the operator with the precedence level of 3 is evaluated first. Operators contained within parentheses are evaluated before other operators, and follow the order of precedence as shown above. Operators at the same level are evaluated left-to-right, except '**' and multiple unary operators on an expression, both of which are evaluated right-to-left. Figure 4-1 illustrates the order of evaluation.

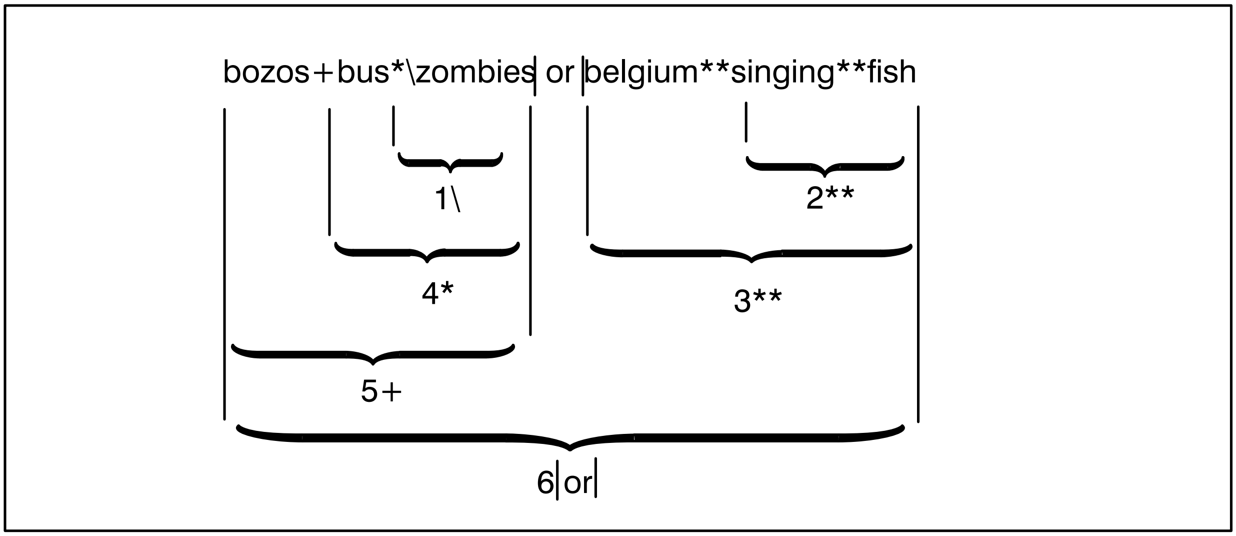

Figure 4-1. Order of Evaluation

Any single alphabetic character inside parentheses is evaluated as a register reference location. Therefore, “`(tp)`” and “`(t+p)`” are legal expressions, but “`(t)`” is not legal.

A numeric value not ending in one of the pre-defined suffixes is evaluated either as a 32-bit integer constant or as a source file line number. A numeric value can be only a line number if it is not expressed in a subscript nor in an arithmetic expression. For example, in “`asparagus(42)`”, the 42 is a constant. Use the ‘`l`’ suffix to force a line number, for example, “`d 2817l+2`”. The “address of” Attribute operator is an exception. For example, “`\204`” evaluates 204 as a line number.

Alternate Variable Display

Debug maintains certain information about each variable in your program. Each location is distinguished by its beginning address, its partition (data, code, EMA), its segment (SEGLD or CDS segment), and the particular bits that contain the object. The bit field contains two parts, the bit offset within the first word at which the object starts and the number of bits the object occupies from that bit position. For example, a normal Pascal integer starts at bit offset 0 of the first word and extends for 32 bits. Debug knows the type of each variable (for example, type integer, real, logical). Whenever you display a variable, the variable is formatted using that type as default.

To override the type, change the bit field characteristics of an object using the type override directives shown below. Whenever a type override is specified, the object starts at bit offset 0 of the first word, the length in bits and the default format are changed to the values listed in Table 4-1.

Table 4-1. Display Types

Override	Bit length	Display format
I	16	Decimal integer
J	32	Decimal integer
O	16	Unsigned octal
B	16	Unsigned binary
Z	16	Unsigned hexadecimal
R	32	Floating point
X	48	Floating point
D	64	Floating point
L	16	FORTRAN logical
C	varies	FORTRAN character, Pascal packed array of char
S	varies	Null-terminated string
A	16	Assembly language

The “repeat count” tells how many adjacent objects of the (possibly overridden) type to display. For example, “d `array[3]\4`” displays elements 3 through 6 of ARRAY. The repeat count usually does not affect the type of the objects and is only significant for the Display command. However, when the C format is given, the repeat count specifies the string length. For example, “d `string\c5`” displays 5 characters, starting with the first word of STRING.

The “display format override” is significant only to the Display command. This overrides the format in which the object(s) are displayed, but does NOT change the “type” of the objects (does not affect the bit offset and bit count characteristics). The objects are extracted according to the type of object (possibly overridden) and displayed in the new format except when the C format is given. When the C format is given with a string size, the length of the object then becomes the specified number of bytes. The following Pascal program is used in the examples below:

```
$debug on$
program Yow;

var
   a:  packed array [1..5] of –128..127:
   b:  packed array [1..5] of set of char;
   i:  integer;

begin
   for i:=1 to 5 do
   a[i] := –i;

   b[3] := [’G’,’I’,’S’];
end.
```

A is an array of integers, packed two to a 16-bit word (8 bits each). Once the END statement is reached, type overrides and display format overrides can be demonstrated as follows.

Debug normally unpacks the elements of A as signed 16-bit integers that are packed into 8 bits. No matter how many bits are actually allocated to a packed object, the object is of a type that requires that many bits rounded up to the next whole word. Pascal and Debug expand the object to occupy entire words when they unpack that object. Consequently, while elements of A are only 8 bits long, Debug unpacks them as the 16-bit integer type.

In the following example, Debug displays each element in its original type, unpacked as necessary:

```
Debug> d a\5
A\5 = -1    -2    -3    -4    -5
```

Entering a display format override does not affect the type of the object. If the format is changed to 16-bit octal, the elements are still unpacked the same way. Using a display format override only affects the way Debug prints the elements.

```
Debug> d a\o5
A\O5 = 177777b  177776b  177775b  177774b  177773b
```

But, if the O type override is specified, the objects become 16 bits (not 8 bits) wide and always start at bit offset 0 of the first word. Debug does not perform any unpacking of the objects. This allows you to view the physical representation of A as actually stored.

```
Debug> d a:O\5
A:O\5 = 177776b    176774b    175400b    0b    0b
```

With no overrides, sets are displayed in set notation with the elements of the appropriate base type:

```
Debug> d b[3]
B[3] = ['G','I','S']
```

However, when the display format override specifies a type that contains fewer words than the type of the object being displayed, the object displays in full. The object B is a set that occupies 17 words. Displaying it with the O (16-bit octal) format override displays 17 words. Note that the first word is an element count word, 256.

```
Debug> d b[3]\o
B[3]\O =     400b    0b    0b    0b    0b
500b    10000b    0b    0b    0b    0b
0b        0b    0b    0b    0b    0b
```

Specifying a display format override that requires more words than the original object causes an error, because nothing can be displayed. For example, displaying the 16-bit type object A (whose elements are packed into 8 bits) as a 32-bit integer results in:

```
Debug> d a[2]\j
D A[2]\J
^
( 63) Format size greater than object size (use type override to change).
```

Substrings

The substring specifier contains the optional starting and ending positions separated by a colon and enclosed in angle brackets. The various forms can be demonstrated by:

< > or <:>	specifies the entire string
<7>	specifies only character 7
<7:>	specifies characters 7 through the current length
<7:10>	specifies characters 7 through 10
<:10>	specifies characters 1 through 10

The values can be between 1 and 32767, allowing character positions past 255 to be displayed and modified.

Substring specifiers can also be used on non-string locations, in which case they act as byte offsets and by default return an 8-bit octal type. For example, if IBUF is an integer array containing the packed ASCII characters "`OF,FST,ID`", then "`d ibuf<4:6>`" displays "`106b 123b 124b`", which is the octal value of characters "`FST`". The command "`d ibuf<4:6>\c`" displays "`FST`".

Special Syntax for Constant Values

Many of the Debug commands give you the option to use either a constant or an expression. If constants are used, they should conform to the syntax described in the following example.

Constants defined as INTEGER and REAL types can take any of the following forms:

`[+|-]`*digits*`[.`*digits*`][E[+|-]`*digits*`]`

`[+|-]`*octalvalue*`B`

`0`*hexvalue*`Z`

where:

digits is a string of 1 to 16 digits.

octalvalue is a string of 1 to 6 octal digits, ending with the letter B.

hexvalue is a string of 1 to 5 hexadecimal digits, beginning with a zero (0) and ending with the letter Z.

Examples:

```
28.1E-7
177777b
0FFFEz
```

Constants defined as COMPLEX types should take the form:

(*realvalue*,*realvalue*)

where:

realvalue follows the same syntax as that defined for integer and real types.

Example:

```
(3.1415,0.007)
```

Constants defined as LOGICAL and BOOLEAN types should take the form:

```
[.]{T|F|TRUE|FALSE}[.]
```

Constants defined as CHARACTER, STRING, and PACKED ARRAY OF CHAR types should take the form:

'*chars*'

where:

chars is a string of characters surrounded by single quotes with enclosed quotes doubled.

Example:

`'Code word: ''SILLY'''` is the string: `Code word: 'SILLY'`

If the second argument fails the above syntax in some way, it is considered an expression and parsed accordingly.

5

Debug Commands

This chapter gives a summary of the Debug commands, their syntax, and a description of each command recognized by Debug. The commands are listed in alphabetical order. Profiling commands also are described in this chapter.

Debug Command Summary

Table 5-1 lists the Debug commands and their description.

Table 5-1. Debug Command Summary

Command		Description
AB	Activate breakpoint	Activates or reactivates a specified breakpoint.
B	Break	Sets a breakpoint at location specified.
BU	Break uplevel	Sets breakpoints at all known return points of the current routine.
CF	Change list file	Changes the file currently being listed to a new file descriptor.
C	Clear	Clears all breakpoints or the breakpoints set at the location specified.
CB	Clear breakpoint	Clears the specified breakpoints.
/	Command stack	Displays a list of Debug commands entered. Allows any of the commands to be selected and executed.
*	Comment	Ignores any input line to Debug that starts with an asterisk (*).
CV	Create Variable	Creates user-defined Debug Session Variable (DSV).
DB	Deactivate breakpoint	Temporarily removes a specified breakpoint without removing the information about it from the breakpoint list.
D	Display	Displays variables.
DA	Display automatically	Displays the values of expressions each time the program stops.
DE	Display entry points	Displays the names of entry points.
DL	Display location	Displays a standard location identifier given an expression.
DS	Display symbols	Displays all local symbol names for a given routine that start with the same characters as a supplied mask.

Table 5-1. Debug Command Summary (continued)

Command		Description
DT	Display type	Gives a brief description of the type of location.
DV	Display variables	Displays a list of the currently defined DSVs.
DW	Display window	Displays the values of expressions in a window.
DO	Do while	Conditionally repeats command lists.
EC	Echo	Prints a message on the terminal and to the log file, if any.
ELSE		Specifies a list of commands to be executed if the condition is not true.
ENDDO		Terminates a DO command.
ENDIF		Terminates an If command.
XQ	Execute	Schedules a program without wait.
EX	Exit	Terminates the program without prompting and exits Debug.
F	Find	Finds a string in the currently displayed source file, starting the search from the current line.
GO	Goto	Moves the current location to be executed to the location specified.
?	Help	Displays a brief summary of the Debug commands.
H	Histogram	Plots a histogram from profiling data.
IF	If	Conditionally specifies an alternative command list.
I	Include	Executes a set of commands from a command file and optionally logs the output to a log file.
KV	Kill variable	Deletes old DSVs from a Debug session.
L	List	Lists a screenful of source code in your program.
M	Modify	Modifies the value of a variable.
O	Overview	Initiates the profiling session.
SS	Operator suspend	Causes Debug to suspend operation until reactivated by the GO system command.
PA	Path	Sets an alternate directory and/or DS node to use when searching for source files to display.
P	Proceed	Allows your program to proceed to the next breakpoint or specified line of source code. A temporary breakpoint is set at location if it is supplied.

Table 5-1. Debug Command Summary (continued)

Command	Description
PT Proceed across terminations	Allows the program to complete and be rescheduled by a parent program.
Proceed uplevel	Sets internal breakpoints at all known return points of the current routine, proceeds and clears the internal breakpoints set.
R Return	Returns from the current breakpoint command file back to the normal command input.
RU Run	Runs a program with wait.
SET Set	Turns specified switches on or off.
?? Status	Displays status information.
S Step	Steps to the next line of source code, or optionally steps into the next subroutine, or steps to the next line of code and traces the values of the specified variables.
T Trace	Shows the location executed without stopping the program.
V View	Changes the number of source lines displayed on the screen.
W Where	Shows the callers of the current subroutine with the names and values of any actual parameters.

Activate Breakpoint (AB)

Purpose: Selectively activates a specified inactive breakpoint.

Syntax: AB [:*count*] *break#* [*break#*...]

count is the new wakeup count for the breakpoints activated.

break# is the number of the breakpoint as shown by the B command.

Description:

This command can be used to reactivate an inactive breakpoint. The breakpoint can be of any type, for example, unconditional or conditional.

Examples:

`Debug> ab 8 9`	Activates breakpoints #8 and #9.
`Debug> ab :3 7`	Activates breakpoint #7 and gives it a wakeup count of 3.

Break (B)

Purpose: Sets a breakpoint (conditional breakpoint, command file breakpoint, command line breakpoint) or displays current breakpoints.

Syntax: `B` [[:*count*] *location* [*condition*|<<*commandfile*|>>*commandline*]]]

count is a wakeup count specifying the number of times this breakpoint should be hit to stop the program.

location is the location to set the breakpoint.

condition is a condition that must be met for the breakpoint to stop the program, of form:

expression operator expression|*constant*

where operator is any of the following:

=	equal to
<>	not equal to
>	greater than
<	less than
<=	less than or equal to
>=	greater than or equal to

For a description of the syntax to be used for constants, refer to the Special Syntax for Constant Values Section in Chapter 4.

commandfile is the name of a file that contains commands to execute when this break is hit.

commandline is a Debug command line to execute when this break is hit.

Description:

This command can be used to display all breakpoints or to set a breakpoint in the program being debugged. There are four types of breakpoints:

- Unconditional
- Conditional
- Command file
- Command line

A conditional breakpoint has a condition attached to it. The program is halted each time it reaches that location, and Debug returns control to the user or Include file if the condition is met. Otherwise, Debug restarts the program.

To display the list of current breakpoints that are set, enter the Break command without any operands. For example:

```
Debug> b
Breakpoints:
# 1 at 78/ZIPPY
# 2 at 80/ZIPPY
       if:  count = 10
# 3 at 80/ZIPPY
       trace:  bobcat\4
```

Debug displays the list of current breakpoints with the number that was assigned for each one when it was set. This number is used to refer to the breakpoint individually by some commands that accept breakpoint numbers as arguments, for example, Clear Breakpoint (CB), or Activate Breakpoint (AB).

To set a breakpoint, enter the Break command following by a location. For example,

```
Debug> b 43
```

Sets a breakpoint at line 43.

To specify a conditional breakpoint, specify the location and the condition to test. For example:

```
Debug> b stac count < 10
```

Sets a breakpoint at routine `STAC` that stops the program whenever variable `COUNT` is less than 10.

To specify a breakpoint using a wakeup count, specify the wakeup count value. For example:

```
Debug> b:5 85
```

Sets a breakpoint that stops the program the fifth time it hits line 85.

or

```
Debug> b:5 stac count > 20
```

Sets a conditional breakpoint that 'sleeps' the first four times hit and functions as a conditional breakpoint after the count is greater than 20.

The count shown by the B command decreases by 1 each time the breakpoint is hit, until it reaches zero and disappears. To change the wakeup count on a breakpoint, use the AB command. For conditional breakpoints, the condition is evaluated only when the break "wakes up."

To specify a command file breakpoint, specify the location and the command file to include at that point. For example,

```
Debug> b 60/q </DUCK/DEMO
```

Sets a breakpoint at line 60 of routine `Q` and begins executing commands from the `/DUCK/DEMO` file. `/DUCK/DEMO` commands continue their execution until the end of the file is reached or a Proceed or Step command is given.

To specify a command line breakpoint, specify the location and the command line to include at that point. For example:

`Debug> b. >> m i i+1;g 222;p`	Sets a command line breakpoint at the current location that increments variable `i`, goes to line 222, and proceeds from there.

For both command file and command line breakpoints, when one angle bracket (‘<’ or ‘>’) is used, the commands read from the file are echoed to the terminal and to the log file (if any). Using two angle brackets (‘<<’ or ‘>>’) suppresses the echo.

A maximum number of 50 breakpoints is allowed. Conditional breakpoints require substantial amounts of time for processing. Therefore, conditional breakpoints should not be set at lines that are executed repetitively unless execution time is not critical.

More than one breakpoint can be set at the same location. When multiple conditional breakpoints are set at the same location, if either condition is true, Debug breaks at the specified location:

`Debug> b stac count < 10`
`Breakpoint set at 70/STAC`
	Break if count is less than 10.

`Debug> b stac count > 20`
`Breakpoint set at 70/STAC`
	Break if count is greater than 20.

`Debug> p`
`At 70/STAC COUNT < 10`
	Proceed to next breakpoint.
	One of the conditions is true (count <10).

For Macro/1000 code in certain situations, the line where a breakpoint is set may be the line following the line specified. When setting breakpoints at an entry point, Debug assumes that you arrive at the memory location by means of a JSB instruction. Because of this, Debug sets a breakpoint at the instruction following the entry point. For example, if the symbol SUBR appears in an ENT statement and it stands for memory location 21042, all JSB instructions would write the return address at location 21042 and continue execution at address 21043. In anticipation of this, Debug sets a breakpoint at location 21043. This means that if the program arrives at location 21042 by means of a JMP instruction or a skip, or by any other means, it will not break until location 21043 is executed.

If line continuation on a statement is used in Macro/1000, the line number that Debug knows for that statement is the last line of the statement.

Break Uplevel (BU)

Purpose: Sets breakpoints at all known return points of the current routine.

Syntax: `BU`

Description:

This command sets breakpoints at all known return points of the current routine, based on non-CDS entry point value or CDS stack frame.

Change List File (CF)

Purpose: Changes the file currently being listed to a new file descriptor.

Syntax: `CF` *filedescriptor*

Description:

This command changes the file currently being listed to the newly specified filedescriptor. If Debug fails to find a file because it has been changed to a new name, the CF command allows you to change to the new name. Debug remembers the new filename for all routines that came from that file.

Example:

If Debug fails to find source file ROCKY.FTN because it has been renamed to BULLWINKLE.FTN, entering

```
    Debug> cf bullwinkle.ftn        Changes to the correct file.
```

Clear (C)

Purpose: Clears all breakpoints or the breakpoint set at the specified location.

Syntax: `C` *location* `|*`

location is the line number, variable name, absolute address, or statement label where the breakpoint or tracepoint was set.

* is an operand that tells Debug to clear all of the breakpoints and tracepoints currently defined.

Examples:

```
    Debug> c 15                       Clear the breakpoint at line 15 in the
                                      currently executing routine.

    Cleared 15/ADDUP

    Debug> c *
    All clear                         This message displays even when there are no
    Debug> _                          breakpoints to clear.
```

Clear Breakpoint (CB)

Purpose: Clears breakpoints or tracepoints that have been set.

Syntax: `CB [`*break#*`] [`*break#*`] [`*break#*`] ...`

break# is the identification number of a breakpoint, 1..50. Successive *break#*s can be repeated to the end of the line.

Examples:

```
    Debug> cb 2                            Clear breakpoint number 2.
    CLEARED BREAKPOINT # 2 AT 80/ZIPPY
    Debug> _
```

Command Stack (/)

Purpose: Displays a list of Debug commands entered. Allows any of the commands to be selected and executed.

Syntax:

```
/[///...]
/ [n]
```

n is an optional number indicating the number of commands to be displayed. Default is up to 24 commands.

Description:

This command is much like the command stack maintained by EDIT/1000 and CI. The last 24 unique commands are listed in the order they were originally entered, with duplicate lines deleted. The cursor is positioned at a blank line at the bottom of the stack, so that the terminal cursor vertical positioning keys can be used to select the command desired. The line selected can then be modified. Pressing the return key causes that line to be read and executed. Positioning the cursor at a blank line and pressing the return key will simply cause Debug to issue another prompt.

Debug uses the value of your $VISUAL environment variable, if defined, to determine which style of command stack editing is preferred.

Command stack commands are illegal in a command file used in the Include command. Command lines less than three characters in length are not saved in the stack (although they are still processed by Debug). This is to prevent loss of the more lengthy commands already in the stack.

The command stack is saved in the debug information file when Debug is exited. Multiple users must have separate copies of the .DBG file.

Comment (*)

Purpose: Annotates Include files.

Syntax: `*` *comment*

Description:

Any command line beginning with an asterisk (*) is treated as a comment line, and no command is executed for that line.

Create Variable (CV)

Purpose: Creates user-defined Debug Session Variables (DSV).

Syntax: `CV` *dsv* `[`*expr*`]`

dsv is the name of a new variable, 1..16 characters.

expr is an expression that becomes the initial value of the DSV.

Description:

This command creates a user-defined variable, called Debug Session Variables (DSVs). These variables are 32-bit integers that exist within Debug and can be displayed and modified much like variables in your program. These variables also remain defined and retain their values from one Debug session to the next, until the program is relinked.

DSVs can be used in the expression syntax wherever a normal location would go, prefixed by an ampersand (&). Only 32 DSVs can be defined at a time. Use the Kill Variable (KV) command to clear old DSVs. Use the Display Variables (DV) command to see all the currently defined DSVs.

Example:

```
Debug> d bitter(11,11)
BITTER(11,11) = 10
Debug> cv kaspritz 5
Debug> cv mongo 11
Debug> m bitter(&mongo,&mongo) &kaspritz
BITTER(&MONGO,&MONGO):  10 => 5
```

Deactivate Breakpoint (DB)

Purpose: Temporarily removes a specified breakpoint without removing the information about it from the breakpoint list.

Syntax: DB *break#* [*break#* ...]

break# is the breakpoint identification number assigned to the break when it was set.

Description:

This command can be used to deactivate an active breakpoint. The breakpoint can be of any type, for example, unconditional or conditional. A deactivated breakpoint will not cause a program to break. However, the breakpoint can be reactivated at any time by using the Activate Breakpoint (AB) command.

Inactivated breakpoints appear in the listing as “inactive at” that location.

Example:

```
    Debug> db 3
    DEACTIVATED TRACEPOINT # 3 AT 80/ZIPPY
```

Display (D)

Purpose: Displays the contents of one or more locations.

Syntax: D [*expression*|<*text_string*> [*expression*] ...]

expression is any expression. A number of different expressions can be specified; as many as can fit in one line of the command can be entered.

text_string is a string of characters between angle brackets (< >). If present, <*text_string*> replaces the usual ‘expression =’ prefix to the following expression.

Description:

Debug remembers the operands that were entered on the previous Display command, and if the command appears with no operand, it displays the variable list previously specified.

Variables declared in segments other than the one that is currently in memory cannot be displayed. An exception to this rule is the SAVE variables as defined in FORTRAN or Macro/1000 programs.

Example:

```
    Debug> d i
    i = 3
    Debug> d <I have been through the loop > i <times.>
    I have been through the loop 3 times.
```

Display Automatically (DA)

Purpose: Displays the values of expressions each time the program stops.

Syntax: `DA [`*expression*`...|!]`

expression is any expression to turn on auto-display.

! is the character to turn off auto-display.

If only “`da`” is entered, auto-display is turned on and Debug displays the previous values of expressions.

Examples:

`Debug> da arf woof`	Turns on auto-display for variable ARF and WOOF.
`Debug> da !`	Turns off auto-display.
`Debug> da`	Turns on auto-display again for ARF and WOOF.

Display Entry Points (DE)

Purpose: Displays the names of entry points.

Syntax: `DE [`*entmask*`|/`*module*`[/`*segment*`]]`

entmask is the name mask Debug uses to search for entry points. It may contain FMP-style wildcard characters “@” and “`-`”.

module is the module name to show all entry points.

segment is the name of an overlay (SEGLD) segment or a string of the form SEG.*nn*, where *nn* is the segment number (seg.0 is the first segment, and so forth.).

Description:

This command displays the names of entry points that pass the entry mask operand. If given, this mask specifies that only entry points that match the mask should be displayed.

Example:

`Debug> de schneider@`
`SCHNEIDER1/SEG1`
`SCHNEIDER2/SEG2`
	Displays the names of all entry points beginning with “SCHNEIDER”.

`Debug> de /pas.degravy`
`PAS.KHYBER`
	Displays the names of all entry points in the first module Debug finds named PAS.DEGRAVY.

Display Location (DL)

Purpose: Displays a standard location identifier given an expression.

Syntax: `DL` *expr*

expr may be any type of expression; for example, location.

Description:

This command displays the standard location identifier for the program location given by the expression. The expression may be of any type.

Example:

```
Debug> dl @160b
16010b/FMPOPEN
```

Displays a standard identifier for the location pointed to by location 160 octal.

Display Symbols (DS)

Purpose: Displays all local symbol names for a given routine that match the supplied mask.

Syntax: `DS [`*symmask*`][/`*routine*`]`

symmask is the symbol name mask to use. It may contain FMP-style wildcard characters “@” and “-”.

routine the name of the routine to search. If not given, the current routine is used.

Description:

This command displays all local symbol names (variables, statement labels) for a given routine that match the supplied mask.

Example:

```
Debug> ds stanford@/university
STANFORD_PARK
```

Displays the names of all symbols starting with ‘`STANFORD`’ in routine ‘`UNIVERSITY`’.

Display Type (DT)

Purpose: Gives a brief description of the type of location.

Syntax: `DT` *location*

Description:

This command gives a brief description of the type for the specified location. Type overrides change the type that Debug displays.

Example:

```
Debug> dt kilgore                  Debug displays the type of KILGORE.
32-BIT INTEGER

Debug> dt trout:i                  Debug always displays type “16-bit integer”.
16-BIT INTEGER
```

Display Variables (DV)

Purpose: Displays a list of the currently defined DSVs.

Syntax: `DV`

Description:

This command displays a list of all of the currently defined Debug Session Variables (DSVs).

Example:

```
Debug> dv
JUGGLING = 3  OMELETTE = 12
```

Display Window (DW)

Purpose: Displays the values of expressions in a window.

Syntax: `DW [`*expression...*`|!]`

`DW` *expression...*	Displays expression in a window.
`DW !`	Turns off display window.
`DW`	Turns on display window with previous expression list.

Description:

This command creates another window on your screen that is updated with the current values of the named expressions. There may be only one display window; entering another expression list erases the old display window and creates a new one.

Example:

```
Debug> dw yelp yipe
```

The display window contains the values of YELP and YIPE.

```
Debug> dw !
```

The display window is turned off.

```
Debug> dw
```

The display window is turned back on with the values of YELP and YIPE.

Do While (DO)

Purpose: Conditionally repeats command lists.

Syntax:

DO *condition*
   command list
ENDDO

condition is a condition that must be met for the command list to continue to be executed, of the form:

expression operator expression | *constant*

where *operator* is any of the following:

=	equal to
<>	not equal to
>	greater than
<	less than
<=	less than or equal to
>=	greater than or equal to

For a description of the syntax to be used for constants, refer to the Special Syntax for Constant Values Section in Chapter 4.

Description:

The DO command is a conditional command that repeats a list of commands until the condition is no longer true. The command list is terminated by the ENDDO command. The DO, command list, and ENDDO must be in an Include file for the command list to be repeated. DO commands can be nested to 16 levels.

Examples:

```
DO tank <> full
   d tank
   s i
ENDDO
```

Repeatedly displays the Pascal enumerated variable ‘`tank`’ and single steps into lines until ‘`tank`’ is equal to the enumeration constant ‘`full`’.

Echo (EC)

Purpose: Prints a message on the terminal and to the log file, if any.

Syntax: `EC string`

Example:

```
Debug> ec Took error exit.
```

Prints the message “`Took error exit.`”.

Else (ELSE)

Purpose: Specifies a list of commands to be executed if the condition is not true.

Syntax: `ELSE`
    command list
`ENDIF`

Description:

The ELSE command is a command used with an IF command. This command specifies a list of commands to be executed if the condition in an IF command is not true. The ELSE command list terminates upon reaching an ENDIF command. The ELSE command cannot be abbreviated.

Example:

```
IF total = 12
    d index
    p
ELSE
    d total
    ex
ENDIF
```

Enddo (ENDDO)

Purpose: Terminates a DO command.

Syntax: `ENDDO`

Description:

The ENDDO command terminates the DO command associated with it. DO commands can be nested to 16 levels, and as such, must have a like number of ENDDO commands.

Example:

```
DO tank < > full
  d tank
  s i
ENDDO
```

Endif (ENDIF)

Purpose: Terminates an IF command or an ELSE command.

Syntax: `ENDIF`

Description:

The ENDIF command terminates the IF command (and any ELSE command) associated with it. IF commands can be nested to 16 levels, and as such, must have a like number of ENDIF commands.

Example:

```
IF total = 12
  d index
  p
ELSE
  d total
  ex
ENDIF
```

Execute (XQ)

Purpose: Schedules a program without wait.

Syntax: `XQ` *prog* [*parameters*]

prog is the name of the program to run.

parameters are parameters to send to the program.

Example:

```
    Debug> xq bbrnr 76
```

Schedules BBRNR without wait, passing the parameter ‘76’.

Exit (EX)

Purpose: Terminates the profiling session or the program being debugged and ends the Debug session.

Syntax: `EX [-R|-M]`

−R Tells Debug to run the program, not abort it, when exiting.

−M Tells Debug to run the program in Debug Monitor Mode, not abort it, when exiting.

Description:

The EX command terminates the current Debug session. The −R option is useful when a child program needs to pass back the correct status to a parent program. When in a Profiling mode, Debug always asks if the overview file should be purged.

Example:

```
    Debug> ex -r
```

Note that if the Debug EX command is entered without the −R or −M option then the child is aborted using a system OF command. This passes back the status value 100000b (−32768) to a waiting parent program, indicating that the child was aborted. If the parent issued an FmpRunProgram call then that call returns FMP error −226, “Program aborted”. If you wish the proper status values that the child specifies via a PRTN or PRTM call to be passed back to the parent, then use the EX −R or EX −M command to terminate the Debug session. This allows the child to run to completion, leaving the returned status value intact.

Find (F)

Purpose: Finds a string in the currently displayed source file, starting the search from the current line.

Syntax: `[`*number*`]F[ind][/|\][`*string*`]`

number is the line number from which to begin the search.

string is the string for which to search.

/ indicates a forward search.

\ indicates a backward search.

Description:

The Find command instructs Debug to search the current source file on display for string. Number can be used to specify what line from which to begin the search. If omitted, the search starts after the current line. A blank can be used in place of the slash to delimit the string; there is no trailing delimiter. Use the List command to get a different file on display, then use the Find command to search that file.

Examples:

```
    Debug> f/varname
```

or

```
    Debug> f varname
```

Searches for `varname` and if found, displays the line on which `varname` appears in the middle of a window on your screen.

```
    Debug> 906f/< >terminationvalue
```

Debug searches for the line containing “`< >terminationvalue`” starting from line 906 in the source.

```
    Debug> f\function
```

Searches backward for a line containing “`function`”.

Goto (GO)

Purpose: Moves the current location for execution to the location specified.

Syntax: `G` *location*

location is a line number, variable name, statement label name, or absolute address.

Description:

The location must be in the segment that your program is executing. The program does not execute any instructions, therefore, the lines between the old current line and the new current line have not been executed. Therefore, to avoid confusion, use this command with caution. It is not recommended practice to go into the middle of a DO loop or into another subroutine.

The location should be qualified with the routine and/or segment name if it is not in the current routine. A line that does not produce executable code or contains code not compiled and linked with the symbolic debug option cannot be specified in the GO command.

Help (?)

Purpose: Displays a brief summary of the Debug commands.

Syntax: `?[`*command*`]`

Description:

The Help command displays a short summary of the Debug commands available. When entered with a command, a brief description of that command is displayed.

Histogram (H)

Purpose: Plots a histogram from profiling data.

Syntax: `H [-w] [<`*subroutine*`>]`

−w causes Debug to ignore time spent in waiting states.

subroutine specifies that the code in the subroutine specified is to be examined and plotted. Default is to plot all the routines in the program.

Description:

The H command examines and processes the data taken as a result of the Overview command, either from the file created by Debug or the file specified in the command string.

If no subroutine name is specified, all routines taking up more than .5% of total execution time will be plotted, starting with the busiest. All routines taking less than .5% of the total time will be grouped in a collective category under “Other (known code)”. Samples that occur in subroutines such as library routines that do not have Debug information are grouped into a category indicated as “Other (unknown code)”.

All time spent in Waiting (non-scheduled) states can be ignored. The −w option plots the histogram as if no time were spent in any state other than scheduled, although system noise is still included.

If (IF)

Purpose: Conditionally specifies an alternative command list.

Syntax:

`IF` *condition*
  command list
`[ELSE` *command list*`]`
`ENDIF`

condition is the condition that must be met for the command list to be executed, of form:

expression operator expression | *constant*

where *operator* is any of the following:

=	equal to
<>	not equal to
>	greater than
<	less than
<=	less than or equal to
>=	greater than or equal to

For a description of the syntax to be used for constants, refer to the Special Syntax for Constant Values Section in Chapter 4.

Description:

The IF command is a conditional command that chooses an alternative command list if a condition is true, or when an ELSE command is also used, chooses a different command list. The IF command takes as an argument a condition identical to the conditions that can accompany breakpoints. This command is intended to be used in Include command files. IF commands can be nested to 16 levels and each IF must be followed by an associated ENDIF command.

Example:

```
IF total = 12                       If variable TOTAL equals 12,
  d index                           display the value of variable INDEX
  p                                 and run to the next breakpoint.
ELSE                                Else,
  d total                           display the value of TOTAL
  ex                                and exit.
ENDIF
```

Include (I)

Purpose: Executes a set of commands from a command file and optionally logs the output to a log file.

Syntax `I [-Q]` <*command file*> `[[+]`<*log file*>`]`

command file is the name of a file containing one or more Debug commands, not including another I command.

log file is an optional output file name. If specified, all Debug outputs will be stored in this file, as well as being displayed at the terminal. If specified, Debug creates the log file if it does not exist. Default is no log file.

+ tells Debug to append to the log file rather than overwrite it.

−Q tells Debug not to echo the commands read from <command file>.

Description:

After opening the command file, Debug executes the commands from that file until the end of the file is reached. At that point, Debug returns to the interactive mode, displaying the Debug> prompt. If an error occurs, Debug reports the error and reads the next line from the file.

To specify only a log file, enter the following:

```
Debug> I 1 logfile
```

This entry specifies that input is from the terminal and LOGFILE is the log file. To terminate logging in this case, enter a CNTL-D (pressing the CNTL and D keys simultaneously).

If a log file is specified, Debug commands and messages output by Debug are logged into this file. This is useful for acquiring extensive logs of trace information, which can be helpful when tracking a subtle problem. The program being debugged cannot write to the log file. These files do not affect the operation of the user program.

Nesting of the I command is not allowed. Command stack operations are also not allowed by Debug in the command file.

Kill Variable (KV)

Purpose: Deletes old DSVs from a Debug session.

Syntax: `KV` *dsvname*

dsvname is the name of the Debug Session Variable to be deleted.

Description:

DSVs remain defined and retain their values from one Debug session to the next, until the program is relinked. Only 32 DSVs may be defined at a time. The Kill Variable command deletes those specified, making room for new variables. Use the Display Variables command to get a list of all defined DSVs.

Example:

```
KV hoopla
```

List (L)

Purpose: Lists a screenful of source code from your program.

Syntax: Any of the following:

```
L
L location
L <line #>  [filename]
L +[<line count>]
L -[<line count>]
L .
L /<list history entry>
```

location is any location.

<line #> [*filename*] lists the specified line number of either the current source (default) or of the optional file named.

+[<*line count*>]
−[<*line count*>] is a number that specifies the number of lines forward (+) backward (−) from the present block of lines displayed. For example, if lines 4 through 18 are displayed, entering "`L -3`" will display lines 1 through 15. Default (only the plus or minus sign entered) displays the next or previous block of lines in the file, with a two-line overlap.

• displays a block of lines centered on the current line (the line to be executed next).

<*list history entry*> is the number of an entry in the location history stack, 1..5. Refer to the Status (??) command description later in this chapter for additional information.

Description:

The L command lists a number of lines. When a breakpoint is encountered, Debug positions the source file of the routine being debugged on the screen in such a way that the line about to be executed next appears in the middle of the display. This line is called the current line and identified with a marker (>). The list command displays a new block of lines according to the operand specified, with the specified line positioned near the center of the block. The default number of lines displayed is 15, which can be changed by the View command.

With the List command, the rules for specifying locations are not strictly enforced. Debug allows you to specify any line number within the source file, including comment lines. If the location specified is an entry point name, then the first executable line of the module is displayed at or near the center of the block of lines listed.

There are six forms of the L command:

- The L command can be used without an operand to scan forward through the source code. If no operands are specified, Debug displays the next block of lines with a two-line overlap.
- The second form of the List command is used to list the lines around the specified location.
- The third form of the List command (with line number and file name specified) can be used to list other source files or text files. If the file name is omitted, the source code of the program being debugged is listed. To list another file, a line number followed by a space, followed by a file name, is required.
- The next two forms of the List command syntax are used to list different parts of the file that currently appear on the screen. The positive or negative line count allows you to go forward or back a number of lines or to the next or previous screenful. For example, "`L -20`" lists a screenful of lines starting 20 lines above where the screen currently is positioned.
- The "`L .`" form of the List command resets the screen back to show the lines around the line you are about to execute. This command is used after you have scanned through the file or files. It can also be used to refresh the screen in the event that your program does some I/O to the screen.
- The last form of the list command syntax lists the location saved in an entry in the listing history stack. See the ?? command for details on this stack. For example, "`L /3`" lists the location that is in entry #3 of the listing history stack.

The List command places Debug in listing mode. In listing mode, pressing carriage return at the command prompt repeats the last L+ or L− command entered, or executes the command L+ if none. For example, "L−1" followed by several carriage returns lists backwards in the file one line at a time. Debug informs you that listing mode has been entered by changing the prompt to "`List>`".

Example:

```
Debug> l 1                       List the file at line number 1.
List> _                          Now carriage returns will continue to list the
                                 file.
List> _
List> d DAY                      A command other than a List command
DAY = FRIDAY                     exits listing mode.
Debug> _
```

Modify (M)

Purpose: Alters the value of a variable.

Syntax: `M` *variable constant*|*expression*

variable is the variable name to be modified.

constant is a constant. For a description of the syntax to use for constants, refer to Special Syntax for Constant Values in Chapter 4.

expression is an expression.

Description:

Debug interprets the value as being the same type as the variable data type. For example, if the command “`M X 23`” is entered, and X is a 4-word floating point number, the operand 23 is formatted as a 4-word floating point number before it is placed into the memory location for X.

You can override the type of the variable by using the type-override operators with the M command. For example:

```
Debug> d x                    The default type of X is integer.
X = 65                        The current value of X is displayed.

Debug> d x:c2                 Display X in character format.
X:C = ' A'

Debug> m x:C 'AB'             The operand is interpreted as a character string.
X:C: ' A' => 'AB'             The quotes must be present. The value of X is
Debug> d x                    modified from string ‘A’ to ‘AB’. Then new
X = 16706                     value of X  is displayed in default format
Debug> _                      (integer).
```

In this example, the number of characters modified is the length of the character string.

If the new value given is a real number, then digits must appear on both sides of the decimal point. The rounding and truncation rules of FORTRAN are followed on such values.

When modifying locations with hexadecimal values, the hexadecimal value must begin with a numeric digit to distinguish it as a numeric value.

```
Debug> m stac FFFFZ           Debug will interpret this as a reference to
M STAC FFFFZ                  a symbol named FFFFZ.
 ^
(17) Symbol not found
Debug> m stac 0FFFFZ          By specifying a leading zero, Debug knows
STAC: 123456b => 177777b      this is a numeric value.
Debug> d stac:z               Display STAC as a hexadecimal number.
STAC:Z = 0FFFFz
Debug> _
```

Operator Suspend (SS)

Purpose: Causes Debug to suspend operation until reactivated by the GO system command.

Syntax: `SS`

Description:

The Operator Suspend command causes Debug to suspend until reactivated by the GO system command.

Overview (O)

Purpose: Initiates the profiling session.

Syntax: `O` [*filename*]

filename specifies the file that contains the profiling data to be accessed by Debug. If omitted, Debug creates a file using a name built according to the file-naming conventions described below.

Description:

This command instructs Debug to begin taking profiling data on your program. If there are breakpoints set, they are automatically cleared.

If the optional file descriptor is specified, Debug does not create a new data file, nor does it run your program to collect new data. It opens the file to obtain the profiling data.

If the program file resides on a CI hierarchical directory, the profile data file name is formed using the name of the program file with a type extension of .OVR. If the program file resides on a FMGR cartridge, an equal sign (=) is used, for example, if the program file name is PROG1, the profile data file name will be =PROG1.

Path (PA)

Purpose: Sets alternate directories and/or DS nodes to use when searching for source files to display.

Syntax: `PA [-k] [`*path path* ...`]`

`-k` if this option is given, any existing path list is cleared before the new paths are added to the list. To just clear the path list, enter "`PA -k`".

path any valid directory path. May also contain DS node information for searching on another system.

Description:

Adds new entries to the list of alternate directory paths to look for source files if not found on the local system in the original directory. The search sequence is (1) as given in the symbol table; (2) the current working directory; (3) the current path (if any). If no parameters are entered, the current path is displayed.

Examples:

```
Debug> pa /wolverine/mitchell/ /romlee/
```

Sets two alternate directories.

```
Debug> pa -k @>2817
```

Sets an alternate DS node, first clearing the old path list.

```
Debug> pa
```

Displays the current path.

Proceed (P)

Purpose: Transfers control to the program being debugged until the next breakpoint is encountered, the program terminates, or a program violation such as an MP or DM error occurs.

Syntax: `[`*repeat*`]P[[:`*count*`]` *location*`[`*condition*`|<<`*commandfile*`|>>`*commandline*`]`

repeat is an optional incremental value from 1 to 32767 that specifies the number of breakpoints Debug should execute before returning control to the user. The default number is 1.

`:`*count* specifies the number of times the temporary breakpoint should be hit before it suspends the program.

location is an optional operand that defines a temporary breakpoint. This means that Debug sets a breakpoint at the specified location, transfers control to your program, stops at that breakpoint, and then clears that breakpoint upon return to Debug.

condition is a condition that must be met for the breakpoint to stop the program, of the form:

expression operator expression|*constant*

where operator is any of the following:

=	equal to
<>	not equal to
>	greater than
<	less than
<=	less than or equal to
>=	greater than or equal to

For a description of the syntax to be used for constants, refer to the Special Syntax for Constant Values section in Chapter 4.

commandfile is the name of a file from which to execute when this break is hit.

commandline is a Debug command line to execute when this break is hit.

Description:

The optional *count* parameter is most useful in loops, where a break is desired after a number of loop iterations. For example:

```
Debug> 40p                       Proceeds for 40 breakpoints.
```

If the location operand is specified, it must be a line with executable code. Otherwise, Debug displays the message “`Bad line number`”. In this example, line 23 does not contain executable code:

```
Debug> p 23
P 23
  ^
Bad line number
Debug> _
```

Proceed Across Terminations (PT)

Purpose: Allows the program to complete and be rescheduled by a father program.

Syntax: [*repeat*]`PT`[[`:`*count*] *location*[*condition*|<<*commandfile*|>>*commandline*]

repeat is an optional incremental value from 1 to 32767 that specifies the number of breakpoints Debug should execute before returning control to the user. The default number is 1.

`:`*count* specifies the number of times the temporary breakpoint should be hit before it suspends the program.

location is an optional operand that defines a temporary breakpoint. This means that Debug sets a breakpoint at the specified location, transfers control to your program, stops at that breakpoint, and then clears that breakpoint upon return to Debug.

condition is a condition that must be met for the breakpoint to stop the program, of form:

```
expression operator expression|constant
```

where operator is any of the following:

=	equal to
<>	not equal to
>	greater than
<	less than
<=	less than or equal to
>=	greater than or equal to

For a description of the syntax to be used for constants, refer to the Special Syntax for Constant Values section in Chapter 4.

commandfile is the name of a file from which to execute commands when this break is hit.

commandline is a Debug command line to execute when this break is hit.

Description:

The PT command allows the program to complete and be rescheduled by a parent program. The command works the same as the Proceed command, but terminations are not trapped. When Debug is signaled that the program has been rescheduled, the message “Initial schedule detected” is printed. Your program is not stopped, but continues to a breakpoint.

Example:

```
Debug> pt 4/main
```

Proceeds to line 4 of routine MAIN, allowing an intervening termination.

Proceed Uplevel (PU)

Purpose: Sets temporary breakpoints at all known return points of the current routine and runs the program.

Syntax: `PU`

Description:

This command sets temporary breakpoints at all known return points of the current routine and runs the program.

Return (R)

Purpose: Returns from the current breakpoint command file back to the normal command input.

Syntax: `R`

Description:

The Return command returns from the current breakpoint command file, back to the normal command input, either to interactive mode or an Include file. If given from an Include file, Debug goes back into interactive mode.

Run (RU)

Purpose: Runs a program with wait.

Syntax: `RU` *program* [*parameters*]

program is the name of the program to run.

parameters any parameters to send to the program.

Description:

The Run command runs a program with wait.

Example:

```
Debug> ru dl @.cmd
directory::HENRY
HEYTHERE.CMD
Debug> _
```

Runs the program DL, and passes the parameter @.cmd to DL. Control returns to Debug after DL finishes execution.

Set (SET)

Purpose: Turns specified switches on or off.

Syntax: `SET [`*switch*` [ON|OFF]]`

switch is any one of the following values:

`PASCAL` If ON, location parsing occurs as normal: carats (^) denote pointer dereferences and periods (.) denote record subfields. If OFF, periods and carats are legal symbol characters. If PASCAL is OFF, all symbols are truncated to 16 characters (for FORTRAN). For example, "`d d.rtr`" with PASCAL ON looks for record 'D' with subfield 'RTR'; with PASCAL OFF, it looks for symbol 'D.RTR'. The default PASCAL setting for programs that contain Pascal routines (with debug info) is ON.

`STEPIN` If ON, the default for single-stepping is to Step Into subroutines (this occurs when just 's' is given or just <return> is pressed at a 'Step>' prompt). The 'Over' modifier steps over calls when STEPIN is ON. For example: 's' steps over calls when STEPIN is ON. The default setting is OFF.

`EXIT` If ON, terminations actually complete (Debug says 'Program ran to completion'). If OFF, all terminations are trapped ('Program attempted to terminate'). The default setting is OFF.

`RECORD` If ON, only the commands read from the Include file (usually the terminal) are logged to the log file, other output is suppressed. This lets you keep a record of commands needed to set up certain debugging situations. The default setting is OFF.

`LOG` If OFF, logging is suspended but the log file is kept open. If set back ON, logging resumes (the new output is appended to the log file). The default setting is ON.

`ML` If ON, Debug operates in machine-level mode; if OFF, it operates in source-level mode. The default is OFF.

`CSC` If ON, all special characters outside the normal ASCII range are converted to octal format in string displays. If OFF, control characters are still converted to octal, but characters with bit 7 set (ASCII values 128..255) are printed without conversion. The default setting is ON.

Description:

The Set command toggles various switches ON or OFF. Just saying SET *switch* toggles the switch between ON and OFF. Just saying SET shows the current settings of all switches. Switches retain their values from one Debug session to the next. The switch names can be abbreviated.

Example:

```
Debug> se exit on
SWITCH EXIT IS NOW ON
```

Status (??)

Purpose: Displays status information.

Syntax: `?? [LH]`

Description:

The Status command, by default, displays the name of the program being debugged and the file it was RPed from (if Debug RPed it). The ?? LH command shows the location history stack, a list of the last five "current locations" listed. This is very useful when your program suddenly jumps to an unexpected location; you may want to examine the code that called for the branching.

Step (S)

Purpose: Allows line-by-line execution of the program being debugged.

Syntax: [*repeat*] `S [I|O] [U|T` *expr expr* ...`]`

repeat is a value indicating how many single steps to execute before returning control to the user. The default is 1. The space between the repeat value and S is optional.

`I` for stepping into a subroutine

`O` for stepping over subroutines

`U` to turn off tracing

`T` for tracing variables

expr specifies any expression to display when the tracepoint is reached.

Description:

This command transfers control to the user program only for execution of the current line or number of lines specified by the repeat value parameter. Then control returns to the user.

Debug displays a new prompt (`Step>`) when the Step command is entered. Subsequently, a carriage return is interpreted as the Step command until any other command is entered.

In the case where the current line to be executed contains calls to subroutines or functions that you would like to debug, the 'I' may be specified. This instructs Debug to step into the subroutine called on that line. By default, the subroutine call would be stepped over, and control returned to the user on the line following the call. For example, assume that a program is currently positioned at the line shown below:

```
.
.
>   234  Call Subr (x,y,z)
.
.
```

Entering the S command would execute subroutine SUBR, and return control at line 235. If any breakpoints were encountered in subroutine SUBR, Debug would break at those breakpoints.

The Step Into command (S I) allows you to single-step into a subroutine. Entering “`s i`” when Debug is halted at line 234 would transfer control to the first executable line of code in subroutine SUBR. You can single-step through this subroutine or enter any other Debug commands. When you step through the Return statement, Debug returns to the main program at line 235.

The Step Into command requires that the subroutine to be executed be compiled with the symbolic debug option (refer to the Compiler Options section in Chapter 2). If the subroutine was not compiled with the debug option, Debug steps over the subroutine call, ignoring the option.

If the subroutine does not use a standard calling sequence and contains debug information, the Debug will either single-step into it or say it cannot be single-stepped. Debug outputs a message to inform you to proceed to the next line.

Another option for the Step command (Step Trace) allows you to trace the values of a list of variables after every step. This is done with the T (Trace) option. For example:

```
    Debug> s t var1 var2
```

instructs Debug to single-step the current line and display the values of the variables `VAR1` and `VAR2` after every single-step. Entering S T (with no variable list) returns you to the previous variable list entered.

The Step command behaves differently in machine-level mode. In machine-level mode, disassembled code is displayed and the Step command steps one machine instruction at a time. However, it is possible to step over subroutine calls by entering “`S O`” for Step Over.

Example:

```
    Debug> s t answer:i (a)       Single-step and trace ANSWER and the
    ANSWER:I = 0  (A) =    0b     A-Register.
    Step> s                       Another Step command repeats the
    ANSWER:I = 0  (A) = 26113b    step trace.
    Step> s u                     Entering the Step Untrace command exits
    at 503/SUM                    step trace mode.
    Step> s t                     Just entering Step Trace uses the
    ANSWER:I = 42 (A) =    0b     previous variable list.
```

Trace (T)

Purpose: Shows the location about to be executed without stopping the program.

Syntax: `T[[:`*count*`]` *location* `[`*expr expr ...*`]]`

`:`*count* specifies the number of times the temporary breakpoint should be hit before it begins displaying values.

location is where the tracepoint is set. It follows the location specification rules.

expr specifies any expression to display when the tracepoint is reached.

Description:

A tracepoint is a special breakpoint that does not cause control to be returned to the user. Instead, when encountering a tracepoint, Debug simply indicates that the tracepoint has been reached by displaying the line number where a tracepoint was set, and allows the program to continue. Using the tracepoint command, you can follow the execution of the program without having too many interruptions.

The Trace command also allows an optional list of variables to be displayed when the tracepoint is executed. If the T command appears with no operand, a list of the current tracepoints that have been set is displayed. Since tracepoints are really special breakpoints, a combined total of 50 breakpoints and tracepoints are allowed.

Set a tracepoint at line 24 of the current subroutine.

```
Debug> t 24
Set Tracepoint # 1 at 24/ADDUP
Debug> p
At 24/ADDUP                     This message is displayed every time line
                                24 is executed.
```

Set a tracepoint at line 20 and display variable FinalTotal.

```
Debug> t 20 FinalTotal
Set Tracepoint # 1 at 24/ADDUP
Debug> t
Tracepoints:
20/ADDUP  Trace:  FINALTOTAL
Debug> p 26
At 20/ADDUP FINALTOTAL = 12
At 20/ADDUP FINALTOTAL = 46
The final total is:  145
Debug> _
```

As with the Break command, the scope of the variables that appear in the Trace command must be accessible to Debug when the tracepoint is executed. You can specify as many variables as you can enter in one command line. It is always a good idea to fully qualify any location.

View (V)

Purpose: Changes the number of source code lines displayed on the terminal or, if no parameters are given, just refreshes the display window.

Syntax: `V [[+|-]` *lines*`]`

`+`	increase the display window by *lines.*
`-`	decrease the display window by *lines.*
lines	the number of lines by which to change the size of the display window.

Description:

The View command changes the number of lines of source code displayed on the terminal. If the number of lines is preceded by a plus (+) or a minus (−), the size will be relative to the existing window size. The number of lines will be changed to an odd number so that the current line can be displayed at the center of the window on the screen. The window is then refreshed. If no new window size is given, the existing window is refreshed.

Example:

`Debug> v + 10`	Increases the number of lines of source code displayed by 10.
`Debug> v`	Refresh existing window.

Where (W)

Purpose: Shows the current line, subroutine, and segment of the program being debugged. Also shows the routines that called the currently executing subroutine.

Syntax: `W [-q][[`*startdepth*`]` *enddepth*`]]`

`−q`	do not display parameter values.
startdepth	is the call chain depth at which the traceback should start. The default is 1.
enddepth	is the call chain depth at which the traceback should end. The default is the end of the call chain.

Description:

The Where command shows the callers of the current subroutine with the names and values of any actual parameters. The output resulting from this command includes several lines displaying the following information:

- First line shows the line number, subroutine name, and segment name of the routine your program is currently executing.
- The next line displays the subroutine that called the current one, and the names and values of the actual parameters that were passed.
- Whenever applicable, there is another line for the caller of that subroutine and the parameters passed. Subsequent lines identify preceding subroutine callers. The output stops when the main program is reached, or the main routine of a segment is reached.

The values of parameters displayed by the Where command represent the values at the time that the command is entered, not at the time the subroutine was called. If the parameter value has changed since the subroutine call occurred, the new value is displayed. The parameters may not be in the same order as they appear in the source code. The names are those used as formal parameters by the called routine.

When your program is stopped in routines that were not compiled with the debug option, or in RTE system library routines, the Where command does not show you the module name. Instead, the octal address of where your program is stopped is displayed. The callers of this unknown routine are not displayed.

Examples:

The following example illustrates use of the Where command. It assumes that the main program `PROG` called `STARTUP` that called `OPENFILES` that in turn called `FILEERROR`. Assume that the program is currently stopped in subroutine `FILEERROR`. The result of the Where command is shown below.

```
Debug> w
At 35/FILEERROR
Callers:
340/OPENFILES
     DCB = 0   ERRORFLAG = .FALSE.
122/STARTUP
     COMMANDBUFFER = 12345
34/PROG
Debug> _
```

The variables `DCB` and `ERRORFLAG` are parameters passed from `OPENFILES` to `FILEERROR`, and `COMMANDBUFFER` is a parameter from `PROG` to `STARTUP`. Array parameters have only the first element displayed.

```
Debug> w 2
```

This shows the name of the current routine and traces back only the last 2 calling routines.

```
Debug> w 10 10
```

This shows the name of the routine at depth 10 in the call chain.

6

Profiling

This chapter describes the profiling capability of Debug/1000. A brief explanation of a program profiler is provided at the beginning of this chapter followed by a description of the Debug profiling features and a sample profiling session. Profiling commands are described in the Debug Commands chapter.

Introduction to a Program Profiler

A profiler shows the relative execution time of subroutines in a program. With this kind of information, you can optimize the parts of code using the most CPU time, and make your program run faster and more efficiently.

In profiling mode, a histogram can be generated that shows at a glance which routines need to be improved. The same can be done for the lines in each routine. Thus, you can determine where your program is executing.

Debug Profiling Features

The profiling capability of Debug is a separate mode from the normal debug mode. You can initiate a debugging session and switch into the profiling mode, but after the switch has been made, you cannot return to the debug mode. You must initiate another debug session for further debugging.

The default operating mode is for debugging. To switch to profiling mode, the Overview command is used.

While in the profiling mode, Debug provides the following features:

- Plots a histogram of the routines in your program.
- Plots a histogram of the lines in a routine.

In performing profiling functions, Debug alternatively runs and waits at 10-millisecond intervals and samples your program's location counter. Because it attributes the entire 10-millisecond timeframe to your program, it is essential that no other programs use this time. Therefore, your system should be quiet while profiling. System noise occurs whenever Debug samples the program and finds that it is preempted by another program.

Sample Profiling Session

A sample session is shown below to familiarize you with profiling. In this example, it is assumed that all modules of the program have been compiled and linked with the symbolic debug option. A normal debug session is started as previously described. The Overview command initiates the profiling session. It instructs Debug to gather profiling data about the program by running the program to completion. If there are breakpoints set, they are automatically cleared. The following messages are displayed on the screen:

```
Debug> 0
Do you wish to enter overview mode? [y]
Putting data in file PROG1.OVR
Taking data now...
       (Any program output is listed here)
...Finished taking data
Profile> _
```

When you give the Overview command, Debug attempts to put the data it collects in a file. The data file name is formed from the program (type 6) file name. If the program file resides on a CI hierarchical directory, a type extension of .OVR is used for the data file. If the program file resides on a FMGR cartridge, an equal sign (=) is used so that if the program file name is PROG1, then the profile data file name will be =PROG1.

If a file with the same name exists, Debug asks for permission to purge it. A “yes” answer purges the existing file, and Debug runs the program and collects new information in the .OVR file. If you answer “no”, Debug does not purge the file or collect new data; Debug uses the data already in the file.

This file can be kept at the end of a profiling session and used later to obtain more histogram plots.

The message “`Taking data now...`” indicates that the program is running and Debug is monitoring it.

During program execution, Debug takes samples of the program location counter every 10 milliseconds and places this data in the data file. Debug counts the time that the program spends waiting on I/O, and the time spent executing.

When the program completes, Debug outputs the message “`...Finished taking data`” and prompts for the next command.

The H command provides information about where the program is spending its time. To continue the sample profiling session:

```
    Profile> h
    Processing profile data...

    Routine               Amount           Histogram

    GIMME                    8%           ***************
    ATOL                     7%           **************
    GETCH                    5%           *********
    WHERESLAVE               5%           *********
    RPEEK                    4%           ********
    IN                       4%           ********
    ADROF                    4%           ********
    CLEARB                   4%           **
    MVW                      1%           **
    LTOA                     1%           **
    SETB                     1%           **
    DOPOK                    1%           **
    DEST                     1%           *
    GTCLK                    1%           *
    WINDX                    1%           *
    SSTEP/DEBU1              1%           *
    MBT                      1%           *
    WHERE                    1%           *
    Other (known code)       5%           *********
    Other (unknown code)    18%           *************************
    I/O suspend             11%           *********************
    Wait state              20%           **************************
    System noise             1%           *
    Profile> _
```

In the sample above, routines executing most often are plotted with the busiest routines at the top. The percentage is rounded up to the nearest whole number and should be used only for comparison.

The names of routines are truncated to 16 characters.

Any routines taking up less than 0.5% of the execution time are grouped together under "`Other (known code)`". These add up to 5% in the sample above.

Routines compiled with ASMB or FTN4 or those that have been modified by OLDRE appear under "`Other (unknown code)`". Most of the system library routines fall under this category.

The "`I/O suspend`" category includes time spent by the program waiting on unbuffered I/O devices like disks.

The "`Wait state`" is time spent while waiting on another program, buffered I/O, or some other waiting conditions. These states are different under RTE-A because there are different types of such states.

To plot a histogram of subroutine ATOL in the sample program:

```
Profile> h atol
Profile for module ATOL

7% of total time spent here.

Line No.         Amount       Histogram

  764             2%          ****
  773             1%          *
  778             1%          *
  783             1%          **
  786             1%          **
  790             1%          *
  802             1%          **
  811             8%          *****************
  812            10%          **********************
  813             7%          ****************
  821             8%          *****************
  822            17%          *******************************
  824             9%          *******************
  827            13%          ****************************
  840             1%          *
  843             1%          *
  848            18%          **********************************
  852             1%          *
  853             1%          **
  868             1%          *
Profile> _
```

In this plot, any line number that took up any execution time and was sampled is listed. Comment lines and lines without any data samples are not listed. Up to 1000 lines can be profiled.

7

Xdb Compatibility Mode

Xdb provides an interface to Symbolic Debug/1000 that is similar to the xdb debugger for HP-UX. Xdb provides a superset of Debug/1000 functionality; all existing Debug/1000 commands are available, as well as the Xdb commands.

Any Debug/1000 command line may be entered by preceding it with a colon (:). Any command not recognized as an Xdb command is passed to the Debug/1000 command handler.

In Xdb mode the term "current location" refers to the location currently listed on the screen, rather than the point of suspension of execution (as it does in Debug).

Xdb Runstring

To run the debugger in Xdb mode, enter the following runstring from CI:

`xdb [-`*options*`]` *program* `[`*parameters*`]`

where:

program is the program you want to debug. It may be specified as a file descriptor of the type-6 file from which the program is to be restored; or the name of an existing program to be adopted in the form "*prog*`[/`*session*`]:IH`".

parameters are the parameters to the program that is being debugged.

options is one or more Xdb options. See the Xdb Runstring Options section that follows for a definition of the available options.

Xdb Runstring Options

Options available when running Xdb are:

−d Add a path to the list of alternate directories where source files are to be searched.

−r Specify a record file.

−p Specify a playback file.

The following subsections define the above options.

Additionally, any Debug/1000 option may be specified. If there is a conflict between a Debug option name and an Xdb option, preceding the option with a dash (−) specifies the Xdb option, and preceding the option with a plus sign (+) specifies the Debug option. Certain Debug options are superceded by Xdb equivalent options listed above and are not listed here.

`+l:`*lu* Redirects Xdb I/O to another terminal LU.
`+v:`*lines* Sets window view size, see the Xdb w command.
`+w` Does not perform initial schedule, waits for another program to schedule the program being debugged.
`+p:`*p1*`:`*p2*`:`*p3*`:`*p4*`:`*p5* Sets the RMPAR parameters explicitly to *p1-p5*.
`+b` Builds the symbol table, but does not initiate a debug session.
`+d` *infofile* Specifies the debug information file name.
`+m` Enters machine-level (disassembly) mode at the primary entry point.
`+rb` Restores the breakpoints from the last session.

−d Option

Purpose: Adds a path to the list of alternate directories where source files are to be found.

Syntax: `-d` *path*

path is the path name.

Description:

A path is added to the list of alternate directories where source files are to be searched for. The *path* parameter may also specify DS file access information; for example, "`-d @>5003`".

−r Option

Purpose: Specifies a record file.

Syntax: `-r` *file*

file is the file name of the record file.

Description:

Specifies a record file. Recording begins to this file immediately (for overwrite, not for append). See the Record and Playback Commands section in this chapter.

−p Option

Purpose: Specifies a playback file.

Syntax: `-p` *file*

file is the file name of the playback file.

Description:

Specifies a playback file. Playback begins from this file immediately. See the section Record and Playback Commands in this chapter.

Command Stacking

Xdb uses the value of your $VISUAL environment variable, if defined, to determine which style of command stack editing is preferred. Because Xdb defines the slash (/) character to be a command, the "CI-style" command stack must be accessed by preceding the command with a colon. For example, if you prefer the CI-style command stack and wish to perform command "`/3`", you must enter "`:/3`".

If file *programname*`.XSTK` exists in the same directory as the .DBG file, Xdb restores the command stack from that file at startup, and saves the command stack to that file when terminating. For instance, before debugging a program whose .DBG file is `/raspberry/cordial.dbg`, you can enter CI command:

```
   CI> cr /raspberry/cordial.xstk
```

Xdb will now save and restore the command stack using this file. This feature may be used to preserve the command stack when your program is relinked. Xdb and Debug always save the command stack in the .DBG file, but that saved stack is lost when the program is relinked.

Xdb Commands

The following sections describe the Xdb commands. First the command arguments are defined, then the Xdb commands themselves are defined.

Command Arguments

The *address*, *depth*, *location*, and *format* arguments can be used in the Xdb commands described in the following section. The command arguments are defined as follows:

address Specifies a code address in much the same format as the normal debug mode. The address is a numeric value suffixed by one of the following characters:

b	value is an octal data partition address
a	value is a decimal data partition address
c	value is an octal code partition address
q	value is an octal Q-relative address
e	value is a 32-bit decimal EMA/VMA address

depth Specifies the integer depth of a desired routine invocation in the call chain, as reported by the Xdb t command.

location Specifies the name of a procedure or the line number of an executable statement in the program. The syntax is:

line
#label
file [*:line*]
[*segment:*] [*file:*] *proc* [*:proc* [...]] [*:line* | *#label*]

where:

line is a source file line number expressed as a decimal integer.

#label is a source language statement label name or number, preceded by a pound sign (#). For C, the label is the identifier named in a labelled statement; for FORTRAN, the label is a statement number; for Pascal, the label is a label number for a labelled statement.

file the name of a file specified in "filename.ext" syntax only, without directory path or any other FMP file descriptor fields.

segment is the five-character name of a non-CDS segment.

proc is the name of a procedure (or a "routine" or "function") in your program. There may be more than one of these specified to indicate Pascal nested procedures.

format For commands that display variables, these specify an alternate format for the display. Note that a different syntax applies when your program is suspended in a routine written in the C language. That syntax is documented in the "C Language Expressions and Format" section of this chapter. The syntax for languages other than C is:

[*count*] *formchar* [*size*]

where:

count is an integer number of times to apply *formchar*. The default is one.

formchar is one of:

`n`	normal
`d`	decimal
`u`	unsigned decimal
`o`	octal
`x`	hex
`z`	binary
`b`	decimal byte
`c`	character
`e`	floating point
`f`	floating point
`g`	floating point
`i`	assembly instruction
`s`	string

size is an integer number of bytes to use, or one of these characters:

`b`	byte
`s`	short
`l`	long
`D`	double

Commands

The Xdb commands are defined in the subsections that follow. They are grouped into the following categories:

Breakpoint commands
Execution control commands
Data display commands
Stack tracing commands
File viewing commands
Window control commands
Record and playback commands
Miscellaneous commands

Table 7-1 is a summary of the Xdb commands.

Breakpoint Commands

These commands are used to set or delete breakpoints. Associated with each breakpoint are three attributes:

location This is the same as the "*location*" command argument defined previously. The breakpoint is "hit" whenever the named location is about to execute.

count This specifies the number of times the breakpoint must be hit before it is recognized. The syntax is:

```
\[n][t | p]
```

where:

n is a number from 1 to 32767 that specifies the count; the default is 1.

`t` specifies that the breakpoint is temporary, that is, will be cleared when recognized.

`p` specifies that the breakpoint is permanent (this is the default).

commands Actions to be taken upon recognition of a breakpoint before waiting for command input. These are separated by ';' and can be enclosed in '{}' to delimit the list saved with the breakpoint from other commands on the same line. If the first character is anything other than '{', or if the matching '}' is missing, the rest of the line is saved with the breakpoint.

Saved commands are not parsed until the breakpoint is recognized. If there are no commands, Xdb waits for command input after recognition of the breakpoint. For immediate continuation, finish the command list with '`c`'.

Xdb has only one active command line at a time. When it begins to execute breakpoint commands, the remainder (if any) of the old command line is discarded with notice given.

Table 7-1. Xdb Command Summary

Breakpoint Commands	
lb	List breakpoints.
b [*location*] [\count] [*cmnds*]	Set breakpoint at *location*, or at the current location if none specified.
ba *address* [\count] [*cmnds*]	Set a breakpoint at the given code address.
bu [*depth*] [\count] [*cmnds*]	Set an up-level breakpoint.
bc *number count*	Change the count associated with a breakpoint.
db [*number*]	Delete the given breakpoint by *number*, or all breakpoints at the current location if a number is not specified.
db *	Delete all breakpoints.
sb [*number*]	Suspend the given breakpoint *number*, or all breakpoints at the current location if a number is not specified.
sb *	Suspend all breakpoints.
ab [*number*]	Activate the given breakpoint number, or all breakpoints at the current location if a number is not specified.
ab *	Activate all breakpoints.
The following are not strictly breakpoint commands, but are most commonly used within breakpoint command lists:	
Q	If the “Quiet” command appears as the first command in a breakpoint command list, the normal announcement of “breakpoint at address” is not made.
L	Display the file name, procedure name, and line number corresponding to the object code being executed or examined.
”*string*”	Print the given string.
Execution Control Commands	
c [*location*] C [*location*]	Continue after a breakpoint. A temporary breakpoint is set at *location*, if given.
s [*count*]	Single-step one statement, or *count* statements, if given.
S [*count*]	Single-step like ‘s’, but step over procedure calls.
Data Display Commands	
p *expr*[\format]	Print the value of an expression.
p *expr* ? *format*	Print the address of an expression (C language only).
Stack Tracing Commands	
t [*depth*]	Trace the call chain to a maximum of *depth* levels (default is 20).
File Viewing Commands	
v [*location*]	View the code at *location*.
V [*depth*]	View the code at which a routine in the call chain is suspended.
va [*address*]	View the code at *address* in the source window.
+ [*lines*]	Scroll the display down by the given number of lines, or down one line if not specified.

Table 7-1. Xdb Command Summary (continued)

File Viewing Commands (continued)	
− [*lines*]	Scroll the display up by the given number of lines, or up one line if not specified.
/[*string*]	Search forward through the current file for *string*, or for the last string found if none supplied.
?[*string*]	Search backward through the current file for *string*, or for the last string found if none supplied.
n	Repeat the last find command issued.
N	Repeat the last find command, but reverse the direction of the search.
D [−k] [*path path ...*]	Add entries to the list of alternate directory paths in which source files are to be searched for.
ld	List the alternate directory paths.
Window Control Commands	
w [+|−] *lines*	Set size of source viewing window.
td	Toggle disassembly mode.
u U	Update (refresh) the display.
Record and Playback Commands	
>*file*	Set or change the record file to *file* and turn recording on.
>>*file*	This is the same as >*file*, but appends to *file* instead of overwriting.
>@*file*	Set or change the record-all file to *file*, for overwriting or appending.
>>@*file*	Same as >>*file*, but appends to *file* instead of overwriting.
>(t | f | c)	Turn recording on (t) or off (f), or close the recording file (c).
>@(t | f | c)	Turn record-all on, off, or close the record-all file.
tr [@]	Toggle record or [record-all].
>	Display the current recording status (same as '>>').
>@	Display the current record-all status (same as '>>@').
<*file*	Start playback from *file*.
<<*file*	Start playback from *file*, using the single-step feature of playback.
Miscellaneous Commands	
q	Quit Xdb.
<*carriage-return*> ~	Repeat the last command, if possible, with an appropriate increment, if any.
![*runstring*]	Execute the given runstring, or run CI if no runstring supplied.
h [*topic*] help [*topic*]	Print commands/syntaxes related to this topic.
I	Print information (inquire) about the state of Xdb.

The breakpoint commands are as follows:

`lb` List breakpoints.

`b` [*location*] [\count] [*commands*]

Set breakpoint at *location*, or at the current location if none specified.

`ba` *address* [\count] [*commands*]

Set a breakpoint at the given code address.

`bu` [*depth*] [\count] [*commands*]

Set an up-level breakpoint. The breakpoint is set immediately after the return to the procedure at the specified stack depth (the default is one, not zero). A depth of zero means "current location", for example "`bu 0`" is a way to set a temporary breakpoint at the next location to be executed.

`bc` *number count* Change the count associated with a breakpoint.

`db` [*number*] Delete the given breakpoint by *number*, or all breakpoints at the current location if *number* is not specified.

`db *` Delete all breakpoints.

`sb` [*number*] Suspend the given breakpoint number, or all breakpoints at the current location if *number* is not specified.

`sb *` Suspend all breakpoints.

`ab` [*number*] Activate the given breakpoint number, or all breakpoints at the current location if *number* is not specified.

`ab *` Activate all breakpoints.

The following are not strictly breakpoint commands, but are most commonly used within breakpoint command lists:

`Q` If the "Quiet" command appears as the first command in a breakpoint command list, the normal message of "breakpoint at address" is not displayed. This allows, for example, quiet checks of variables to be made without cluttering up the screen with unwanted output. The 'Q' command is ignored if it appears anywhere else.

`L` Display the file name, procedure name, and line number corresponding to the object code being executed or examined. This is most useful in assertion and breakpoint command lists.

"*string*" Print the given string. This command is useful for labelling output from breakpoint commands.

Execution Control Commands

`c` [*location*]
`C` [*location*] Continue after a breakpoint. A temporary breakpoint is set at *location* if given.

`s` [*count*] Single-step one statement, or *count* statements if given. Procedure calls are stepped into. Successive carriage returns will repeat with a count of one.

`S` [*count*] Single-step like ‘`s`’, but step over procedure calls.

Data Display Commands

`p` *expr*[\format] Print the value of an expression. The syntax of this command is different when your program is positioned in a routine written in the C language. For languages other than C the expression is specified in Debug’s non-Xdb-mode syntax; also, the print command does not perform variable assignments. To modify a variable in those languages, use the Debug Modify command. (Refer to the “C Language Expressions and Formats” section in this chapter for information on the syntax accepted for C routines.)

`p` *expr* `?` *format* Print the address of an expression. This command may be used only when your program is suspended in a routine written in the C language.

Stack Tracing Commands

`t` [*depth*] Trace the call chain, to a maximum of *depth* levels (default 20).

File Viewing Commands

`v` [*location*] (lowercase ‘v’) View the code at *location*. If no location is given, scroll the current listing down a page.

`V` [*depth*] (uppercase ‘V’) View the code at which a routine in the call chain is suspended. The code is either the call to the next routine in the call chain, or the next statement to be executed for depth 0 (the default).

`va` [*address*] View the code at *address* in the source window. This command is most useful when in disassembly mode.

`+` [*lines*] Scroll the display down by the given number of lines, or down one line if not specified.

`-` [*lines*] Scroll the display up by the given number of lines, or up one line if not specified.

`/`[*string*] Search forward through the current file for *string*, or for the last string found if none supplied.

`?`[*string*] Search backward through the current file for *string*, or for the last string found if none supplied.

`n` Repeat the last find command issued.

`N` Repeat the last find command, but reverse the direction of the search.

`D [-k] [`*path path* `...]`

Add entries to the list of alternate directory paths in which source files are to be searched for. Each path may include DS file access information, for example:

```
D /altdir/@>altnode
```

If '`-k`' is given, the existing path list is cleared before the new entries are added. To simply clear the path list, enter "`D -k`".

`ld` List the alternate directory paths.

Window Control Commands

`w [+|-]` *lines* Set size of the source viewing window. If the value is preceded by a '`+`' or '`-`', the size is set relative to the existing window size.

`td` Toggle disassembly mode. When in source mode, the source window is redrawn to display the code in assembly language. In addition, the single step command steps one assembly instruction at a time rather than one source statement at a time. The assembly language display consists of the address, the contents of the word in octal, the assembly instruction, and the entry point name if the word is an entry point or the source line number if the word is the first word of a source line. If the debugger is already in disassembly mode, the td command causes the source window to be redrawn to again display program source and line numbers.

`u` (or `U`) Update (refresh) the display.

Record and Playback Commands

The debugger supports a record/playback feature to help re-create program states and to record all debug output. This is particularly useful for debugging that requires a lengthy setup. There are two ways to record debug output:

record only commands are recorded to the record/playback file.

record-all all debug output is copied to the record/playback file including prompts, commands entered, and command output. However, child process output is not captured.

Once the commands have been recorded to the record file, it can then be used as a playback file. Each command line from the playback file is presented before it is executed. A menu allows you to execute (<cr>) or skip (`S`) the line, execute more than one line (<*num*>), continue without prompting (`C`), quit (`Q`) single stepping, or ask for help (`?`).

Note that a record-all file cannot be used as the playback file. The record/playback file name cannot be '`t`', '`f`', or '`c`', and cannot begin with an '`@`'.

There is a significant difference in the implementation of these features between Xdb/1000 and the xdb for HP-UX: Xdb/1000 does not implement separate files for "record" and "record-all". Turning on one of these features closes the file associated with the other, if any.

The following commands are available:

`>`*file* — Set or change the record file to *file* and turn recording on. This rewrites *file* from the start. Only commands are recorded to this file.

`>>`*file* — This is the same as `>`*file*, but appends to *file* instead of overwriting.

`>@`*file* — Set or change the record-all file to *file*, for overwriting or appending. All debugger output is copied to the record-all file, including prompts, commands entered, and command output.

`>>@`*file* — This is the same as `>@`*file*, but appends to *file* instead of overwriting.

`>(t | f | c)` — Turn recording on (`t`) or off (`f`), or close the recording file (`c`). When recording is resumed, new commands are appended to the file. In this context, '`>>`' is equivalent to '`>`'.

`>@(t | f | c)` — Turn record-all on, off, or close the record-all file. In this context, "`>>@`" is equivalent to "`>@`".

`tr [@]` — Toggle recording; if ON turn it OFF, if OFF turn it ON. If '@' is specified, toggle record-all.

`>` — Display the current recording status (same as "`>>`").

`>@` — Display the current record-all status (same as "`>>@`").

`<`*file* — Start playback from file.

`<<`*file* — Start playback from file, using the single-step feature of playback. Each command line from the playback file is presented before it is executed. A simple menu lets you execute (<cr>) or skip (`S`) the line, execute more than one line (<*num*>), continue without prompting (`C`), quit (`Q`) single stepping, or ask for help (`?`).

Only command lines read from the keyboard or a playback file are recorded in the recordfile. For example, if recording is turned on in an assertion, it does not "take effect" until assertion execution stops.

Command lines beginning with '`>`', '`<`', or '`!`' are not copied to a record file (but they are copied to a record-all file). To override this, begin such lines with blanks.

Miscellaneous Commands

`q` — Quit Xdb.

<carriage-return>
`~` (tilde) — Repeat the last command, if possible, with an appropriate increment, if any. Repeatable commands are those that print a line, print a window of lines, print a data value, single step, and single step over procedures. *<carriage-return>* is saved in a record file as a `~` (tilde) command.

![*runstring*] Execute the given runstring, or run CI if no runstring supplied.

h [*topic*]

help [*topic*] Print commands/syntaxes related to this topic using the CALLS utility. The help command without a topic prints a command summary. Available topics include command names, for which a description of the command and the syntax will be printed. Use the command "h @" to get a list of the other available topics.

I Print information (inquire) about the state of Xdb.

C Language Expressions and Formats

When your program is positioned in C code, the expressions entered as arguments to the p (print) command are specified in a syntax that is a subset of the full HP C/1000 syntax. All C operators except the following are provided:

function()	Function call
,	Evaluate left side and discard result
?:	Conditional expression evaluation

No operations are defined for the "pointer to function" type. If displayed, the value of the first word is shown in octal.

The logical AND (&&) and logical OR (||) operators always evaluate both operands. This is an incompatibility with C/1000. In C/1000, these operators always evaluate the left operand first, and then evaluate the right operand only if necessary. That is, if the value of the left operand is such that the relation can not return true, then the right operand is not evaluated.

Type Names and Specifiers

Cast operations may be performed only to integer, floating point, char, and void types, and to pointers to and arrays of those types. Casts may not be performed to function, struct, union, or enumeration types, or to named types given in a typedef. The same rules apply to the argument of the "sizeof" operator.

The allowed syntax for these <type-name> entries is:

```
<type-name> ::=
  <type-specifier><abstract-declarator>

<type-specifier> ::=
  char
  short
  int
  long
  unsigned
  unsigned char
  unsigned int
  unsigned long
  float
  double
  void
```

<abstract-declarator> ::=
<empty>
* *<abstract-declarator>*
^ *<abstract-declarator>*
far * *<abstract-declarator>*
<abstract-declarator> [<expression>]

The syntax for *<type-specifier>* omits these grammatical elements available in C/1000:

<struct-or-union-specifier>
<enum-specifier>
<typedef-name>

The syntax for *<abstract-declarator>* omits these:

(<abstract-declarator>)
<abstract-declarator> ()
<abstract-declarator> (<parameter-type-list>opt)

Also notice that the *<expression>* between square brackets in the *<abstract-declarator>* syntax is not optional. It can be any general expression, not just a *<constant-expression>* as in C/1000.

Special Symbols

Xdb provides a number of special “variable names” that allow access to hardware registers and to variables that reflect the status of the debugging session. Each of these special variable names are preceded by a dollar sign. These symbols may be used anywhere a *<primary-expression>* is appropriate in C expression syntax. The symbols are:

$a The A (accumulator) register. Equivalent to data address 0.

$b The B (accumulator) register. Equivalent to data address 1.

$x The X (index) register.

$y The Y (index) register.

$e The E (extend) register.

$o The O (overflow) register.

$c The C (CDS on/off) register.

$q The Q (CDS stack pointer) register.

$cq The C and Q registers together, C in bit 15.

$pc The PC (program counter) register.

$map0
:
$map31 Program mapping registers for pages 0 through 31. These may be displayed with a repeat count to access sequential maps, for example, “`p $map0\32o`”. Equivalent to data addresses 10b–47b.

Each of these symbols is considered to name an lvalued “int”, that is, a 16-bit integer that may be modified, as with an assignment operator. For example, to display the A register in octal, use “`p $a\o`”; to modify the X-Register to 42, use “`p $x=42`”.

Special Operators

Xdb provides a number of special operator names beginning with a dollar sign that may be used in expressions anywhere that a <*primary-expression*> is appropriate (although they behave like <*unary-expression*>*s)*. The operators are:

`$addr` <*cast-expression*>

Return the address of the lvalued <*cast-expression*>. The type of the result is “pointer to type T”, where T is the type of the <*cast-expression*>. This is functionally identical to the C “&” unary operator.

Example: “`$addr(upc)`” returns the address of variable “upc”.

`$sizeof` <*unary-expression*>
`$sizeof` *(* <*type-name*> *)*

Return the size of the operand in bytes as an unsigned int. This is functionally identical to the C “sizeof” operator.

Example: “`$sizeof(servo)`” returns the size of variable “servo”.

`$in` <*location*>

Return 1 if the <*location*> names the procedure in which the program is currently suspended, else return 0. The result is an “int”.

Example: “`$in monkey.c: print_scopes`” returns 1 if the program is suspended in procedure“print_scopes” of file “monkey.c”.

Display Formats

The optional format argument to the p (print) command instructs Xdb how to display the expression (or the address of the expression). The syntax is:

[count] formchar [size]

where:

count is an integer number of times to apply formchar. If the type associated with the combination of formchar and size implies a size smaller than the size of the expression result, and if count is not specified, then Xdb will use a default count that applies formchar enough times to display the entire result.

formchar is one of:

a expression names a null-terminated string.

b decimal byte (equivalent to “db”).

c character byte.

d, D decimal integer (1, 2, or 4 bytes).

e, E floating point in exponent form (4 or 8 bytes).

f, F floating point in decimal fraction form (4 or 8 bytes). Exponent form is used if the fraction has more than 10 leading zeros.

g, G floating point in fractional form if the fraction has 5 or fewer leading zeros, else print in exponent form (4 or 8 bytes). This is the default for float and double types.

i assembly instruction (2 bytes).

k, K formatted dump of structure (same as “n”).

n normal (default) format, based on result type.

u, U unsigned decimal integer (1, 2, or 4 bytes).

o, O octal integer (1, 2, or 4 bytes).

s treat expression as a pointer to a null-terminated string; display the string pointed to.

S formatted dump of structure (same as “n”)

x, X hexadecimal integer (1, 2, or 4 bytes)

z, Z binary integer (1, 2, or 4 bytes)

If an uppercase version of the formchar is listed together with the lowercase version above, the uppercase version implies a “long” (4-byte) type for integers, or a “double” (8-byte) type for reals.

If the lowercase version is used without an explicit size given, Xdb chooses among the available byte sizes for the format according to the size of the expression result. For instance, displaying a char, int, or long with format “\x” will display in hexadecimal as a 1-, 2-, or 4-byte type, respectively.

size is an integer number of bytes to use for the format type, or one of the following characters:

b byte (1 byte)
s short (2 bytes)
l long (4 bytes)
D double (8 bytes)

If a numeric value is specified then it must be either 1, 2, 4, or 8.

If the object size specified by the format is greater than the size of the expression result and if the result is “lvalued” (that is, names an object in memory that may appear to the left of an assignment operator), then it is expanded in size to read enough bytes from memory to satisfy the format type. For example, if an int is displayed with format “\d4” or “\D”, then Xdb behaves as if the result names a long integer, and reads 4 bytes from memory. If the expression result is instead “rvalued” (that is, names a constant value), then an attempt to use a format greater than the rvalued object size generates an error message.

If the type associated with the combination of formchar and size implies a size smaller than the size of the expression result, and if count is not specified, then Xdb uses a default count that applies formchar enough times to display the entire result. For example, if a long int is displayed with format “\o1”, then Xdb behaves as if the format were entered as “\4o1”, displaying 4 bytes in octal. Similarly, displaying a struct variable with format “\x2” displays all the words of the struct in 16-bit hex.

A

Error Messages

This appendix provides a list of Debug error messages in numerical order, some of which are self-explanatory. These messages are in the error message file /CATALOGS/DEBUG.C<langid>.

The numbers in this list are in sequential order. Most of the missing numbers are used for messages, and are not errors. Others are for errors that should never occur, but if they do appear, represent a problem in Debug and should be reported to your HP representative.

1 Address out of range.

Debug tried to access the location specified, but found that a DM or MP violation would occur.

2 Only 50 breakpoints allowed.

This includes the breakpoints Debug automatically sets while single-stepping.

3 Bad value.

The value specified is out of the correct range for this command.

5 Schedule error: <message>

The program given in the runstring or run command could not be scheduled. Details may appear in <message>.

6 Unknown command.

Debug cannot recognize the command entered.

7 Too many symbols defined to this routine.

8 Unsupported constant type.

An expression names a constant symbol of a type that Debug cannot display.

10 Duplicate entry point:

These are duplicate entry points or module names found in the symbol table file for the program produced by LINK. Change the source to have unique names, recompile, and relink.

11 No breakpoint there.

There is no breakpoint at the location specified to the Clear command. Specify a location with a breakpoint or use CB to clear a particular breakpoint.

12 Bad line number.

The line specified is not appropriate for this command. The line number must be within the current routine or must be qualified by specifying a routine name or segment name. It must be an executable line.

13 Variable must be in the same segment as the breakpoint.

Conditional breakpoints and tracepoints require that the variable accessed be in the main or the same segment as the breakpoint or tracepoint. For CDS programs, the variable must be in either the data partition, or the same segment as the breakpoint or tracepoint.

14 Integer expected.

A single integer value is expected.

15 Location is not a pointer.

The up arrow (^) dereference was found on a non-pointer location.

16 Illegal character.

17 Symbol not found.

18 Location expected.

Xdb expected a “location” to be entered as an argument to the command.

19 Expecting =, < >, <=, >=, > or <.

An invalid condition comparison operation was found.

20 **No debug info for this module: <modulename>**

Debug is suspended inside a module that does not have debug information. If the module name is unknown, '??' is given.

21 **Location must be in current segment or main.**

Only variables in the main or the currently loaded segment can be displayed or modified.

22 **Illegal use of EMA address.**

A breakpoint cannot be set at an EMA address nor can an EMA address be listed.

23 **Invalid display format override.**

24 **Value doesn't match variable's type.**

The value given is of a type not compatible with that of the variable on the left.

26 **Invalid address.**

Xdb expected an "address" command argument, but the argument supplied was invalid.

29 **Unknown data type.**

The type of the object specified is not known to Debug.

30 Only = and < > legal with this type.

Only comparisons for equal and not equal are allowed for complex data types.

31 Legal keywords are Into, Over, Trace, and Untrace.

An illegal Step command keyword was found.

32 Can't single-step. Please use breakpoints.

Debug is unable to single-step this line. This may occur when there is privileged code in one of the routines called by this line or special micro-code has been encountered. Set a breakpoint at the next location. To get the program past this point, use the P command.

33 Value too small.

34 Value too large.

35 No INCLUDE nesting.

An Include command cannot be given from an Include file.

36 File name expected.

37 Caller not available.

The Where command could not determine the caller of this routine due to multiple entry points. Debug cannot determine which entry point was used.

38 The run string must begin with 'RU,': <runstring>

Debug's runstring must be in format "RU,DEBUG,<parameters>".

41 This pointer is NIL. Doesn't point to anything.

An attempt has been made to dereference a Pascal pointer that is nil.

42 Maximum subscript nesting level exceeded.

Subscripts can only be nested 10 levels deep.

45 Legal registers are (P), (A), (B), (X), (Y), (E), (O), (Q), or (Z).

51 Location must be a subroutine name.

54 Only 10 levels of nested procedure definitions allowed.

55 More expected.

A location or subexpression was expected.

56 Caller can't be found (wasn't compiled with debug option).

The Where command did not find any debug information for the caller of the routine. Either the caller wasn't compiled with the debug option or the return address is incorrect.

58 Not enough subscripts. Filling in lower bounds for those not given.

More subscripts were declared for this array than were specified. Debug will use the lower bound for those not given.

59 Too many subscripts given. Ignoring extra ones.

Fewer subscripts were declared for this array than were specified. Debug ignores the extra ones.

63 Format size greater than object size.

An attempt was made to display an object in a format which requires more words than the size of the object, for instance, displaying an INTEGER*2 in INTEGER*4 (\J) format.

64 Display count must be between 1 and 32767.

66 No profile data has been taken; see the Overview command.

67 This routine is not currently active. Locals/Parameters not available.

Variables that are locals or parameters to CDS routines are allocated from the CDS stack while the routine is active (that is, while the routine is in the call chain). These variables may not be accessed when the routine is inactive.

68 Warning: Program was relinked without the debug option.

The .RUN file for the program has been updated after the .DBG file was last updated. This may indicate that the information in the .DBG file no longer matches the .RUN file. This does not necessarily indicate an error.

79 Display LU is not a terminal.

The LU number supplied with the −L option is not a terminal.

82 Program not found.

The program name used with the ':IH' option does not exist.

85 Current module unknown.

93 Warning: Only processing 255 characters.

Debug can modify only up to 255 characters of a string object. Any characters past the 255th will not be modified.

94 Can’t find main program’s debug info.

Debug cannot find the symbol table entry for the main program. You may have compiled the program with an old version of Pascal. Recompile using at least the 2401 version of Pascal.

96 Can’t find starting line. Was main compiled with the debug option?

Debug could not find the starting line for the main program. Recompile using the Debug option.

99 Invalid EMA usage detected.

Debug does not support entry points in EMA.

104 Couldn’t get any data.

The profiler in Debug could not get any data for overview mode. Your program may not have executed long enough for Debug to obtain any timing samples.

108 No data points taken here.

The routine specified in the Histogram command had no data points.

116 Not enough memory.

Debug does not have enough free memory available to run. The program or data segment should be sized up to the full 32 pages.

117 Substring out of range.

An illegal value was given in a substring operator.

118 Symbol table format changed. Please re-load your program.

Your program's debug information file was updated by an obsolete version of Debug.

119 Bad debug information file as output by LINK.

Your program's debug information file was corrupted. Relink.

121 Object size cannot be greater than 128 words.

A user-defined object greater than 128 words was accessed.

123 Too many user-defined types.

Debug encountered too many symbols for types of variables. Recompile some modules without the Debug option.

124 Invalid or uninitialized string.

The string variable accessed had an uninitialized header.

125 Inconsistent number of global user-defined types found.

The Pascal directive $Private_Types$ needs to be declared in subprograms which define global types not declared in the main. See the Pascal manual.

134 No Debug information for that location.

136 You cannot modify the program counter. See the Goto command.

139 The symbol table has already been built.

The −B option was specified but the symbol table had already been built. Debug terminates.

140 Please clear the breakpoint at <location>.

A breakpoint has been set on an instruction that is unknown to Debug. Clear or deactivate the breakpoint at the appropriate location.

143 Object types are incompatible.

The left and right sides of a modify operation are of incompatible types. You may need to use type overrides to force the types to be compatible.

146 Warning: Reached internal breakpoint limit.

Debug may not be able to properly single step the current line because it requires more breakpoints to be set than are currently available. If you are using a large percentage of the maximum number of breakpoints (currently 50), then clearing some of those breakpoints may allow Debug to set all of its internal breakpoints.

148 Maximum nesting level exceeded.

This applies only to Pascal programs.

151 Input file not specified after option.

"−I" was specified, but no input file was given.

152 Log file not specified after −I option.

"−I,<input file>" was specified, but no log file was given.

153 Program name not specified.

No program name was given after the option string.

154 Must not be on base page.

The Goto and Break commands cannot be given base page arguments.

155 Code address must be in current CDS code segment.

A reference was made to a CDS segment other than the current segment.

156 Stack marker inconsistency detected.

Debug found an error in the linking of stack frames while traversing the call chain.

158 Location is not a record.

A record subfield delimiter (.) was found on a non-record location.

159 Subfield not found.

The subfield name given was not defined for the enclosing record type.

160 Warning: missing type information detected.

This represents an error in the symbol table or debug information produced by the compiler. Debug cannot find the type definition for a user-defined object, and will treat it as an untyped machine word.

161 This routine is not at the specified stack depth.

A CDS stack depth level was specified in a qualifier, but the qualifying routine does not exist at that stack depth in the call chain.

162 Maximum IF nesting level exceeded.

IF and DOWHILE commands can only be nested to 16 levels.

163 Incorrect nesting level.

An ELSE, ENDIF, or ENDDO was found when no IF or DOWHILE was active.

164 ID segment gone.

The ID segment for the program has been destroyed; the program has disappeared.

165 Cannot adopt program: insufficient capability.

You do not have sufficient capability to adopt the program in question, such as a program in another session.

166 Cannot adopt program: system DEBUG table is full.

Too many programs are being debugged at once. The number of concurrent Debug sessions is fixed at generation time.

167 Cannot adopt program: could not obtain the required resource number.

No Resource Numbers are free to use for Debug Event signaling.

168 Cannot adopt program: This program is already being debugged.

Only one debugger may adopt a particular program (ID segment) at a time. Another copy of the program can be RPed and debugged if desired.

169 Error running program:

An error occurred while attempting to execute the program, which is explained in the error message returned.

170 Address out of range (MP/DM).

An access to the program was found to violate the memory protection system.

171 I/O error: disk Exec call aborted!

An attempt to access the swap file or type-6 file for the program was aborted, possibly due to a locked LU or a disk device error.

172 Address out of range (past end of program).

An access to the program was found to be beyond the address space defined for the program.

173 VMA access error (could not justify the PTE).

An error occurred while accessing VMA.

174 VMA access failed.

Debug could not create a VMA page fault for the requested VMA page due to locked VMA pages or disk errors on the backing store, such as "`Ran out of disk space`". This can occur if the Working Set size is the minimum of 2 pages.

175 Resource number EXEC call failed.

An internal error occurred on a Resource Number call. This should never happen.

176 Internal synchronization error: maps not set up.

An access was made to the program without required synchronization. This should not happen.

177 Code partition is overlaid, cannot be modified.

An attempt was made to modify the CDS code partition while it was not in memory.

178 Cannot access VMA because program is suspended inside VMA.

The program was adopted while executing the VMA fault handler (VMA). VMA access is prohibited until the program exits VMA.

218 Can’t find return points.

The BreakUplevel or ProceedUplevel command cannot determine the exit addresses of the current routine due to a lack of debug information or confusion caused by multiple entry points, or if the routine looks like a main (has no return address).

219 Invalid breakpoint number.

The breakpoint number supplied to the ClearBreak, DeactivateBreak, or ActivateBreak is out of range or not in use.

220 Command valid only in Include or breakpoint command files.

The Return command is not meaningful for interactive use.

221 ‘+’ or ‘−’ expected.

An option delimiter was expected.

222 Debug information filename not given after −d option.

Nothing appeared after the ‘−d’ runstring option.

224 Missing enumeration constant or record subfield detected.

This occurs when type definition information is found missing while building the symbol table.

225 Invalid switch name.

A parameter to the set (SE) command was unrecognized.

227 Illegal construct (PASCAL is OFF).

A record subfield or pointer dereference was found while the PASCAL switch was OFF, making these references illegal.

232 Expression operands must be 1- or 2-word objects.

Arithmetic operations are only defined for 16-bit and 32-bit integer objects.

235 Arithmetic error: Undefined operation.

An arithmetic operation resulted in a UN math library error (the operation requested is undefined).

236 Arithmetic error: Overflow.

An arithmetic operation resulted in an OF math library error (the operation produced numeric overflow).

237 Arithmetic error.

An arithmetic error other than UN or OF occurred (unlikely).

238 Unary operator expected.

A binary operator was used in unary format (preceding an operand).

239 Binary operator expected.

A unary operator was used in binary format (between operands).

240 Undefined DSV.

Reference was made to an undefined Debug Session Variable.

242 Illegal DSV name.

Debug Session Variable names must be alphabetic strings up to 16 characters.

244 Override size is larger than the expression given.

A type-override/repeat-count combination required more words than are contained in the expression type (2 words), such as ‘d &chance:j2’, which attempts to display two 2-word values.

248 Display formats are not legal here, try type overrides.

Display format overrides may come only at the end of expressions, because they override the final display format.

249 This command is illegal when code is on display.

The Find command does not work in Machine-Level mode.

251 Must be a CDS program to use this feature.

A CDS-only feature was used on a non-CDS program, such as the |LABL| operator.

256 CDS stack depth levels are valid only for stack-relative variables.

A CDS stack depth level was specified for a location that is not a CDS local.

257 Qualifiers illegal on this Location type.

A qualifier was used where invalid, such as on a register.

258 Segment name is illegal here, routine name expected.

Only procedure names can be qualified by segment, not line numbers nor variable names.

269 List window is empty.

A “list forward” was attempted, but there was nothing in the list window.

271 Maximum record nesting level exceeded.

A Pascal record type definition contains too many nested record type definitions for Debug to build the debug information for it.

272 Maximum procedure nesting level exceeded.

A Pascal lower-level procedure is nested at too deep a level for Debug to build the debug information for it.

273 Missing type information detected.

An expected type definition does not appear in the debug information file.

275 Unexpected enumeration constant or record subfield: <symbol>

While building the symbol table, Debug encountered an enumeration constant or record/struct/union subfield symbol that was not expected. This represents an internal error in either Debug or the compiler that produced the debug information.

286 Duplicate symbol: <symbol>

While building the symbol table, Debug encountered a duplicate symbol definition. Debug issues a message indicating that it renamed the symbol to a unique name.

7100 Invalid "$" special name

The Xdb expression evaluator did not recognize the name of a special variable preceded by a dollar sign.

7101 Invalid map number for "$map"; must be 0 to 31

A "$map" special variable was entered, but the map number in the name was not in the range of 0 to 31.

7102 Improper operand type

The Xdb expression evaluator was given an illegal operator/operand combination. An operator is not defined for an operand specified. See the appropriate language manual section on expressions for more information.

7103 Incompatible pointer types in assignment

The Xdb expression evaluator could not assign the right side of an assignment operation to the left side because the pointer types are incompatible. You may need to use a cast operation to cast the right side to the proper pointer type. See the appropriate language manual section on expressions for more information.

7104 Invalid cast operation; types are incompatible

The Xdb C expression evaluator could not cast an expression to the specified type because no cast operation is defined that converts expressions between those types. See the C manual section on expressions for more information.

7105 Invalid use of subscripts on non-array/non-pointer object

The Xdb expression evaluator found subscripts on a subexpression not typed as an array or C pointer. See the appropriate language manual section on expressions for more information.

7106 Bitwise operators cannot be used on operands of type “double”

The Xdb C expression evaluator cannot perform bitwise operations on subexpressions of type “double”. See the C manual section on expressions for more information.

7107 Operation not defined for pointer types

The Xdb expression evaluator found an attempt to use a subexpression of a pointer type with an operator that cannot be used with pointers. See the appropriate language manual section on expressions for more information.

7108 Invalid combination of types for binary operation

The Xdb expression evaluator found an inappropriate combination of operand types used with a binary operator. See the appropriate language manual section on expressions for more information.

7109 Missing operand; argument stack underflow

The Xdb expression evaluator expected an operand to be present to match an operator specified, but no matching operand is found. This can be caused by other evaluation errors.

7110 Procedure name expected after "$in"

The Xdb expression evaluator found a "$in" operator but no procedure name to evaluate following the operator.

7111 Operand must be lvalued or an address constant (array name)

The Xdb C expression evaluator found an operator that requires an lvalue or array name as an operand, but a non-array rvalue was given instead. See the C manual section on expressions for more information.

7112 Length not allowed with “n” format

The Xdb “n” (normal) display format does not allow a “size” parameter to be specified.

7113 Operand must be lvalued

The Xdb C expression evaluator found an rvalued operand matched with an operator that requires an lvalued argument. See the C manual section on expressions for more information.

Other Error Messages

Other error messages that may occur during your Debug session and their explanations are listed below.

DEBUG.C000::CATALOGS is outdated, date code of nnnnnn is required.

The date code of DEBUG.COOO (stored internally) does not match the date code required by the version of Debug executed.

File not found: <name>.DBG.

Debug could not find the debug information file. Recompile and relink with the symbolic debug option.

XXXX Violation at yyyyy.

Your program has caused a memory protect (MP), dynamic memory (DM), or some other violation. XXXX is the type of error (for example, MP or DM), and yyyyy is the octal address or symbolic location where the error occurred.

Because your program was being debugged, the operating system did not abort your program. At this point in the debugging session, you cannot proceed and execute your program without modifying the instruction that caused the violation. However, you can examine memory locations and list lines of code to try and find out why your program caused the violation.

If Debug gives you a line number and a subroutine name for the place the violation occurred, then this line of code contains the instructions that caused the violation. If Debug gives you an octal address for the location of the violation, the violation occurred in some code that was not compiled with the debug option. In this case, it is necessary to consult a load map to find out in which routine the violation occurred.

Index